High-level Compiler Analysis for OpenMP

Sara Royuela Alcázar

PhD Dissertation

Doctoral programme on Computers Architecture

Technical University of Catalonia

April, 2018

High-level Compiler Analysis for OpenMP

Sara Royuela Alcázar

A dissertation submitted to the Department of Computer Architecture at Technical University of Catalonia in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Supervisor:
Prof. Xavier Martorell Bofill, Universitat Politècnica de Catalunya

Dissertation Pre-defence Committee:
Prof. Jesús Labarta Mancho, Universitat Politècnica de Catalunya
Prof. Daniel Jiménez González, Universitat Politècnica de Catalunya
Prof. Josep Llosa Espuny, Universitat Politècnica de Catalunya

External Reviewers:
Prof. Edmond Schonberg, New York University
Dr. Michael Klemm, Intel Corporation

Dissertation Defence Committee:
Prof. Edmond Schonberg, New York University
Dr. Michael Klemm, Intel Corporation
Prof. Jesús Labarta Mancho, Universitat Politècnica de Catalunya

Prof. Daniel Jiménez González, Universitat Politècnica de Catalunya
Dr. Harald Servat, Intel Corporation

Barcelona, April 2018

Abstract

Nowadays, applications from dissimilar domains, such as High-Performance Computing (HPC) and high-integrity systems, require levels of performance that can only be achieved by means of sophisticated heterogeneous architectures. However, the complex nature of such architectures hinders the production of efficient code at acceptable levels of time and cost. Moreover, the need for exploiting parallelism adds complications of its own (e.g., deadlocks, race conditions,...). In this context, compiler analysis is fundamental for optimizing parallel programs. There is however a trade-off between complexity and profit: low complexity analyses (e.g., reaching definitions) provide information that may be insufficient for many relevant transformations, and complex analyses based on mathematical representations (e.g., polyhedral model) give accurate results at a high computational cost.

A range of parallel programming models providing different levels of programmability, performance and portability enable the exploitation of current architectures. However, OpenMP has proved many advantages over its competitors: 1) it delivers levels of performance comparable to highly tunable models such as CUDA and MPI, and better robustness than low level libraries such as Pthreads; 2) the extensions included in the latest specification meet the characteristics of current heterogeneous architectures (i.e., the coupling of a host processor to one or more accelerators, and the capability of expressing fine-grained, both structured and unstructured, and highly-dynamic task parallelism); 3) OpenMP is widely implemented by several chip (e.g., Kalray MPPA, Intel) and compiler (e.g., GNU, Intel) vendors; and 4) although currently the model lacks resiliency and reliability mechanisms, many works, including this thesis, pursue their introduction in the specification.

This thesis addresses the study of compiler analysis techniques for OpenMP with two main purposes: 1) enhance the programmability and reliability of OpenMP, and 2) prove OpenMP as a suitable model to exploit parallelism in safety-critical domains. Particularly, the thesis focuses on the tasking model because it offers the flexibility to tackle the parallelization of algorithms with load imbalance, recursiveness and uncountable loop based kernels. Additionally, current works have proved the time-predictability of this model, shortening the distance towards its introduction in safety-critical domains.

To enable the analysis of applications using the OpenMP tasking model, the first contribution of this thesis is the extension of a set of classic compiler techniques with support for OpenMP.

As a basis for including reliability mechanisms, the second contribution consists of the development of a series of algorithms to statically detect situations involving OpenMP tasks, which may lead to a loss of performance, non-deterministic results or run-time failures.

A well-known problem of parallel processing related to compilers is the static scheduling of a

program represented by a directed graph. Although the literature is extensive in static scheduling techniques, the work related to the generation of the task graph at compile-time is very scant. Compilers are limited by the knowledge they can extract, which depends on the application and the programming model. The third contribution of this thesis is the generation of a predicated task dependency graph for OpenMP that can be interpreted by the runtime in such a way that the cost of solving dependences is reduced to the minimum.

With the previous contributions as a basis for determining the functional safety of OpenMP, the final contribution of this thesis is the adaptation of OpenMP to the safety-critical domain considering two directions: 1) indicating how OpenMP can be safely used in such a domain, and 2) integrating OpenMP into Ada, a language widely used in the safety-critical domain.

Keywords: Compiler analysis, OpenMP, task dependency graph, correctness, safety-critical

Acknowledgements

To the two persons who have been essential in the development of this thesis. Alex, I will never be too grateful to you, for you gave me the very first chance to start my research career, and you provided me with the tools and environments I needed to develop myself. And Edu, I would never have finished without you. You trusted me and believed what I was doing was worth it, and this was all I needed.

Dario, your support, not only professionally, but of course personally, has been priceless. It does not matter if we live apart, if stress gets us grumpy, or if the future is uncertain. In all the twists and turns, you are my rock, and indeed an intrinsic part of this thesis.

My dear colleagues and friends. Roger, you taught me many of the things I know about compilers. Working with you was a challenge that allowed me to learn from an incredible engineer. Diego, sharing with you our very first years as researchers has been an honor. I really miss you and I hope our lives come closer again sometime.

A special mention to my advisor, Xavi, who allowed me to develop all my ideas, and supported (and bore) me all these years. Thanks for your infinite patience and compliance. And also a mention to Miguel, who's knowledge about Ada and accessibility have been essential to complete the last part of this thesis.

Last but not least, mum, dad and my dear Quim. You are the reasons why I am here, or anywhere else. You help me, understand me, bear me and love me no matter what. You are invaluable to me.

Thank you all for coming along with me in this wonderful journey.

The people mentioned in this acknowledgment are (in order of appearance): Alejandro Duran, Eduardo Quiñones, Dario Garcia, Roger Ferrer, Diego Caballero, Xavier Martorell, Luis Miguel Pinho, Carmen Alcázar, José María Royuela, Quim Garcia.

This work has been partly supported by:

- The P-SOCRATES European project (FP7-ICT-2013-10).
- The Severo Ochoa Program (grant SEV-2011-00067), awarded by the Spanish Government.
- The Spanish Ministry of Science and Innovation (under contracts TIN2012-34557 and TIN2015-65316-P).
- The Departament d'Innovació, Universitats i Empresa of the Generalitat de Catalunya (under contracts MPEXPAR - Models de Programació i Entorns Parallels - and 2014-SGR-1051).
- The U.S. Department of Energy by Lawrence Livermore National Laboratory (under Contract DE-AC52-07NA27344).

Contents

Cover **i**

Abstract **v**

Acknowledgements **vii**

Contents **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Goals of this thesis 2

1.3 Contributions 3

1.4 Document organization 4

2 Background **5**

2.1 Programming models 5

2.1.1 OpenMP 5

2.1.1.1 The execution model 6

2.1.1.2 The memory model 8

2.1.2 OmpSs 9

2.1.2.1 The execution model 9

2.1.2.2 The memory model 10

2.1.3 Ada 10

2.1.3.1 Concurrency model 11

2.1.3.2 Safety 14

2.2 Execution environment 15

2.2.1 The Mercurium source-to-source compiler 15

2.2.1.1 Intermediate representation 16

2.2.1.2 Compiler phases 18

2.2.2 The *libgomp* runtime library 19

2.3 Architectures 20

2.3.1 HPC architectures: Intel Xeon 20

2.3.2 Real-time embedded architectures: the Kalray MPPA® processor 22

3 Compiler analysis for OpenMP **25**
3.1 Internal Representation of the code 25
3.2 Classic analysis adapted to OpenMP 26
3.2.1 The Parallel Control Flow Graph 26
3.2.1.1 Tasks synchronization data-flow algorithm 28
3.2.2 Use-Definition analysis 32
3.2.3 Liveness 34
3.2.4 Reaching definitions 35
3.3 Impact 36
3.4 Conclusion 37

4 Correctness in OpenMP **39**
4.1 Contributions of the M.S. thesis 39
4.1.1 Automatic scope of variables 39
4.1.2 Automatic detection of task dependences 40
4.2 Related work 40
4.3 Automatic solution of common mistakes involving OpenMP tasks 41
4.3.1 Variables' storage 41
4.3.2 Data-race conditions 43
4.3.3 Dependences among non-sibling tasks 45
4.3.4 Incoherent data-sharing 46
4.3.5 Incoherent task dependences 48
4.4 Evaluation of the correctness tool 49
4.4.1 Usefulness 49
4.4.2 Comparison with other frameworks: Oracle Solaris Sudio 12.3 51
4.5 Impact 51
4.6 Conclusion 52

5 A Static Task Dependency Graph for OpenMP **55**
5.1 Applicability 55
5.2 Related work 56
5.3 Compiler analysis 57
5.3.1 Control and data flow analysis 58
5.3.2 Task expansion 59
5.3.3 Missing information when deriving the TDG 60
5.3.4 Communication with the runtime 61
5.3.5 Complexity 61
5.4 Runtime support 61
5.5 Evaluation 62
5.5.1 Experimental setup 62
5.5.2 Performance speed-up and memory usage 63

5.5.3 Impact of missing information when expanding the TDG 64
5.6 Impact 65
5.7 Conclusions 65

6 Towards a Functional Safe OpenMP 67
6.1 Is OpenMP a suitable candidate for critical real-time systems? 67
6.2 The OpenMP specification from a safety-critical perspective 68
6.2.1 Related work 68
6.2.2 OpenMP hazards for real-time embedded systems 69
6.2.2.1 Unspecified behavior 69
6.2.2.2 Deadlocks 71
6.2.2.3 Data race conditions 72
6.2.2.4 Cancellation 72
6.2.2.5 Other features to consider 73
6.2.3 Adapting the OpenMP specification to the real-time domain 75
6.2.3.1 Changes to the specification 75
6.2.3.2 Automatic definition of the contracts of a safety-critical OpenMP library 78
6.2.3.3 Implementation considerations 79
6.2.4 Conclusion 80
6.3 Application of OpenMP to a safe language: Ada 81
6.3.1 Related work 81
6.3.2 Analysis of the Ada and OpenMP parallel models 82
6.3.2.1 Forms of parallelism 82
6.3.2.2 Execution model 84
6.3.2.3 Use of resources 85
6.3.2.4 Memory model 85
6.3.2.5 Safety 86
6.3.3 Supporting the Ada parallel model with OpenMP 87
6.3.3.1 Preemption 87
6.3.3.2 Progression Model 87
6.3.3.3 Fork-join Model 88
6.3.4 Supporting the OpenMP Tasking Model in Ada 89
6.3.5 Evaluation 89
6.3.5.1 Experimental setup 89
6.3.5.2 Structured parallelism: Ada parallel model, Ada tasks and Paraffin 90
6.3.5.3 Unstructured parallelism: Ada parallel model and OpenMP task dependences 91
6.3.5.4 Performance benefit of OpenMP: Ada vs. C 91
6.3.5.5 Interplay of Ada and OpenMP runtimes 93
6.3.6 Managing persistent tasks 93

6.3.7 Conclusion . . . 94
6.4 Correctness for Ada/OpenMP . . . 94
6.4.1 Related work . . . 95
6.4.2 Compiler analysis for mixed Ada/OpenMP programs . . . 95
6.4.2.1 Concurrency in mixed Ada/OpenMP programs . . . 97
6.4.2.2 Representation of an Ada/OpenMP program . . . 98
6.4.2.3 Correctness analysis . . . 99
6.4.2.4 Extending the approach . . . 101
6.4.3 Conclusion . . . 102
6.5 Impact . . . 102
6.6 Conclusion . . . 102

7 Discussion **105**
7.1 Conclusion . . . 105
7.2 Impact . . . 106
7.2.1 European projects . . . 106
7.2.2 Programming models . . . 107
7.2.3 Other thesis . . . 107
7.3 Future work . . . 108
7.4 Publications . . . 109

Bibliography **111**

Figures **123**

Tables **125**

Listings **127**

Algorithms **131**

A Diagrams **133**
A.1 Ada task states and transitions . . . 133

B Benchmark Source Codes **135**
B.1 Benchmarks for correctness checking in OpenMP . . . 135
B.1.1 Fibonacci . . . 135
B.1.2 Dot product . . . 137
B.1.3 Matrix multiplication . . . 138
B.1.4 Pi . . . 140
B.1.5 Sudoku solver . . . 141
B.2 Benchmarks for the OpenMP integration into Ada . . . 143
B.2.1 Cholesky decomposition . . . 143

B.2.1.1 C . . . 143
B.2.1.2 C + OpenMP . . . 143
B.2.1.3 Ada . . . 145
B.2.1.4 Ada + OpenMP . . . 145
B.2.1.5 Ada tasks . . . 147
B.2.1.6 Ada + Paraffin . . . 150
B.2.2 LU factorization . . . 151
B.2.2.1 C . . . 151
B.2.2.2 C + OpenMP . . . 151
B.2.2.3 Ada . . . 152
B.2.2.4 Ada + OpenMP . . . 153
B.2.2.5 Ada tasks . . . 155
B.2.2.6 Ada + Paraffin . . . 159
B.2.3 Matrix . . . 160
B.2.3.1 C . . . 160
B.2.3.2 C + OpenMP . . . 160
B.2.3.3 Ada . . . 161
B.2.3.4 Ada + OpenMP . . . 161
B.2.3.5 Ada tasks . . . 162
B.2.3.6 Ada + Paraffin . . . 163
B.2.4 Synthetic: Ada tasks + OpenMP tasks . . . 164

C Acronyms 167

1

Introduction

This thesis has been developed within the scope of the Open Multi-Processing (OpenMP) programming model and high-level compiler analysis techniques. This introductory chapter explains first the motivation that led us to develop this work, then the goals of this thesis as well as its contributions, and finally the remainder of this document.

1.1 Motivation

Current applications in dissimilar domains such as HPC (e.g., simulation, modeling, deep learning, etc.) and safety-critical systems (e.g., autonomous driving, avionics, etc.) require high levels of performance that can only be achieved by means of multi-core devices with different kinds of accelerators, such as many-cores, GPUs (Graphic Processing Unit) or FPGAs (Field Programmable Gate Array). However, boosting performance is not only in the hands of hardware. Parallel programming models are indeed of paramount importance to leverage the inherent parallelism of the devices. That said, the success of a multi-core platform relies on its productivity, which combines performance, programmability and portability. With such a goal, multitude of programming models coexist and, as a result, there is a noticeable need to unify programming models for many-cores [154].

In that context, OpenMP has proved to have many advantages over its competitors. On one hand, different evaluations demonstrate that OpenMP delivers performance and efficiency levels comparable to highly tunable models such as Threading Building Blocks (TBB) [76], CUDA [83], Open Computing Language (OpenCL) [140], and Message Passing Interface (MPI) [79]. On the other hand, OpenMP has different advantages over low level libraries such as Pthreads [106]: a) it offers robustness without sacrificing performance [81], and b) it does not lock the software to a specific number of threads. Furthermore, OpenMP code can be easily compiled as a single-threaded application, thus easing debugging. As a result, OpenMP has emerged as a de facto standard for shared-memory systems by virtue of its benefits: portability, scalability and programmability; in brief, productivity. Furthermore, the extensions included in the latest specification meet the needs of current heterogeneous architectures: a) the coupling of a main host processor to one or more accelerators, where highly-parallel code kernels can be offloaded for improved performance/power consumption; and b) the capability of expressing fine-grained, both

structured and unstructured, and highly-dynamic task parallelism. Besides, the model is widely implemented by several chip (e.g., TI Keystone [144], Kalray MPPA [42] and STM P2012 [28]) and compiler vendors (e.g., GNU [60], Intel [69], and IBM [66]), thus easing portability.

On the other hand, the importance of compilers has increased alongside the development of multiprocessors. Writing correct and efficient parallel code is hard because concurrency gives rise to a number of problems that are nonexistent in sequential programming: race conditions, deadlocks and livelocks, synchronizations, memory partitioning, load balancing, starvation, etc. In this context, compilers are the keystone to achieve performance by means of static analysis and optimization techniques. For that reason, a multitude of techniques has been developed over the years. Nonetheless, the number of specific analyses for OpenMP in general, and the OpenMP task-parallel model in particular, are very scant. Thus, there is a need to develop mechanisms that enable the analysis of OpenMP applications.

Beyond the HPC domain, the importance of program analysis may become crucial for different reasons: limited resources (e.g., embedded systems), system restrictions (critical systems), etc. Such environments, specially critical embedded systems, impose several constraints focused in two areas: functional safety and time-predictability [77]. For this reason, safety-critical systems are commonly developed with programming languages where concepts as safety and reliability are inherent to the language, such as Ada [23]. The characteristics that have made Ada a widespread language in safety-critical domains are the analyzability (e.g., minimizing data coupling across modules), the real-time support (e.g., allowing the specification of restrictions on the features that will be used) and the concurrency capabilities.

But the most advanced safety-critical systems, such as autonomous driving, include applications typical from the HPC domain (e.g., image recognition, sensor fusion, neural networks, etc.). These applications are pushing the introduction of parallel capabilities in current embedded architectures [137] and programming languages [147]. The integration of such capabilities is however troublesome because of the tight restrictions of such systems, and hence efforts must be in the direction of ensuring these restrictions are preserved.

Overall, there is a clear need to converge the HPC and the safety-critical domains. Based on this necessity, programming models need to fulfill the requirements of both environments: high productivity and safety (including correctness and time-predictability). In our opinion, the problem needs to be tackled from the two perspectives. On one hand, the components used in the HPC domain must be adapted to the necessities of the safety-critical domain. On the other hand, the safety-critical domain must include support for such new components.

1.2 Goals of this thesis

In the context of this thesis we define the following goals:

1. Establish a compiler analysis infrastructure with support for OpenMP, specially focused in the tasking model, to serve as a solid basis for building more complex and specific techniques.
2. Extend the compiler analysis techniques dedicated to evaluate functional safety in OpenMP codes using the tasking model.

3. Implement compiler support for the static generation of a Task Dependency Graph (TDG) based on OpenMP tasks dependences. With this, enable the use of OpenMP in systems with low memory resources such as embedded systems, and also enhance the performance in HPC.
4. Exploit the benefits of HPC in the safety-critical domain while preserving the tight requirements of such systems. In this context, consider OpenMP as a suitable candidate to exploit fine-grain parallelism in Ada, as well as develop analysis techniques for ensuring functional safety in combined Ada and OpenMP applications.

1.3 Contributions

In this thesis we present several contributions in the field of the OpenMP programming model considering three aspects: the specification of OpenMP, compiler analysis techniques and runtime development. The analysis and results presented in this thesis are also applicable to the OmpSs programming model, for that reason, although the rest of the document only refers to OpenMP, the reader can imply we allude to OmpSs as well. The contributions are listed as follows:

1. *Compiler analysis infrastructure.* This thesis introduces an analysis framework developed in the Mercurium source-to-source compiler. This includes the development of a set of classic compiler analyses such as control-flow graph, use-definition, liveness, and reaching definitions analyses. This work also extends the previously mentioned analyses with support for the OpenMP tasking model.
2. *Compiler analysis for OpenMP correctness.* Based on the previous contribution, this thesis includes a set of high-level analyses that allow the detection of situations involving OpenMP tasks that may lead to run-time errors, non-deterministic results or loss of performance. This work identifies a set of cases that users should be aware of, and implements in the Mercurium compiler the techniques that allow supplying hints about errors that may occur at run-time for the presented cases. The usefulness of this work is evaluated using different groups of students, and the information provided by the compiler is compared to that obtained with Solaris Sudio to evaluate its quality.
3. *Compile-time generation of a TDG.* Also based on the compiler analysis infrastructure, this thesis includes the implementation in the Mercurium compiler of a new phase that computes a TDG based on the tasks of an OpenMP program. We use the support implemented in the libgomp runtime library in the frame of the P-SOCRATES European project [121] to evaluate the performance of a static TDG against of that of generating the graph at run-time. This evaluation shows the benefits obtained in memory usage, and thus the feasibility of using OpenMP in embedded systems with limited resources.
4. *Integration of OpenMP into safety-critical environments.* The previous works derive in the exploration of OpenMP for domains where correctness and memory bounds are crucial, such as safety-critical real-time systems. The issue is approached from three different perspectives:
   (a) *An OpenMP specification for safety-critical domains.* This thesis analyzes the specification of OpenMP in pursuit of features that can be a hazard in safety-critical systems, and proposes solutions for those features.

(b) *Integration of OpenMP into a safe language: Ada.* This thesis also builds up the introduction of OpenMP into Ada to exploit fine-grain parallelism. In this regard, the thesis covers all the pillars of the integration:

i. *language syntax*: proposal of a new syntax for introducing OpenMP directives in Ada.

ii. *compiler analysis*: development of new compiler analysis techniques specific for Ada and OpenMP to detect race conditions in situations were the Ada concurrent model and OpenMP interplay, as well as in pure Ada applications.

iii. *runtime support*: study of the compatibility between the Ada and the OpenMP runtimes considering the use of OpenMP to implement the Ada tasklet model[1] to exploit structured parallelism, and the use of OpenMP directives in Ada to exploit unstructured parallelism.

A thorough analysis of the Ada and the OpenMP programming models, including execution model, memory model and safety, drives this integration.

(c) *An analysis technique for functional safety in Ada applications using OpenMP*. Finally, this thesis presents an analysis technique aimed at detecting data-race conditions in applications combining Ada and OpenMP. This is meant to be the starting point for a series of techniques that are to cover all issues that concern functional safety in Ada OpenMP applications.

1.4 Document organization

The rest of this document is organized as follows: Chapter 2 expounds the background of this thesis, including different programming models (OpenMP, OmpSs and Ada), software components (the Mercurium source-to-source compiler and the libgomp runtime library for OpenMP), and architectures (an Intel Xeon-based supercomputer and, the Kalray MPPA real-time embedded multiprocessor). Chapter 3 describes the analysis infrastructure for OpenMP implemented in Mercurium. Chapter 4 introduces the analysis techniques implemented in Mercurium for detecting functional correctness issues in OpenMP codes using tasks, as well as the evaluation of this work. Chapter 5 explains the generation of a static TDG based on OpenMP tasks dependences, and evaluates its use in the Kalray MPPA embedded system. Chapter 6 addresses the integration of OpenMP in safety-critical real-time embedded systems from three perspectives: the OpenMP specification, the safe language Ada (considering the three pillars of the integration: language syntax, compiler analysis and runtime support), and the analysis techniques required for applications combining both languages. Chapter 7 presents the conclusions and future work, and Appendix B illustrates the most important parts of the benchmarks used in the development and evaluation of this thesis.

[1] The Ada tasklet model is a fine-grain model of parallelism which is currently under discussion, and is to be part of the Ada202X standard. See details in Section 6.4.

2

Background

This chapter introduces relevant information about the key components used during the development of this thesis. First, the programming models: OpenMP, OmpSs and Ada. Second, the execution environment, involving the Mercurium compiler and the libgomp runtime library. And last, the architectures; particularly, an HPC machine composed by Intel Xeon processors, and a real-time embedded architecture, the MPPA Kalray processor.

2.1 Programming models

From the vast amount of programming models that allow expressing parallelism, we have focused on two. On one hand, OpenMP, because it is a widely spread parallel programming model, with broad support from chip and compiler vendors, and has proved many benefits to obtain productivity. We take advantage of the similarity between OpenMP and OmpSs, and also use OmpSs applications to evaluate some of our work. On the other hand, Ada, a concurrent programming model commonly used in critical domains by virtue of its reliability. We introduce the main characteristics of all three languages below.

2.1.1 OpenMP

OpenMP (Open Multi Processing) is a standard Application Program Interface (API) for defining multi-threaded programs. The main purpose of the language is to provide programmers with a simple yet complete and flexible platform to develop parallel applications with C/C++ and Fortran.

OpenMP is based on high-level compiler directives, library calls and environment variables, and relies on compiler and runtime support to process and implement its functionalities. The language is built around systems where multiple concurrent threads have access to a shared-memory space. A relaxed-consistency memory model describes the visibility of each thread for a given variable. This visibility, defined by means of data-sharing attributes, may be *shared* among threads or *private* to a specific thread or *team* of threads. Furthermore, a fork-join execution model defines where threads are spawned and joined based on the directives inserted by the programmer.

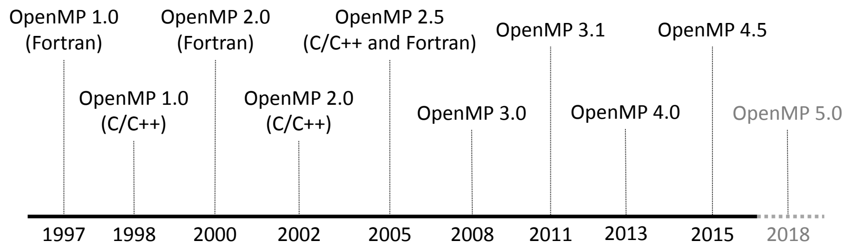


Figure 2.1: Time-line of the OpenMP releases.

Initial versions of OpenMP, up to version 2.5 [108], implemented a *thread-centric model* of parallelism that defines a conceptual abstraction of user-level threads. These conceptual threads work as proxies for physical processors, and thus is a model somehow aware of the underlying resources. The `parallel` and a series of worksharing constructs allow creating and distributing computational work. On account of all that, this model enforces a rather structured parallelism.

The following two releases, versions 3.0 [109] and 3.1 [110], introduced support for a *task-centric model* (also called *tasking model*) of parallelism. This model is oblivious of the physical layout, and programmers focus on exposing parallelism rather than mapping parallelism to threads. As a result, the language allows defining unstructured and highly dynamic parallelism.

The latest versions of OpenMP, versions 4.0 [111] and 4.5 [113], include support for accelerators, error handling, thread affinity and SIMD extensions, expanding the language beyond its traditional boundaries. Furthermore, the specification includes improvements to the tasking model, such as task dependences, and augmentations of the allowed reduction operations, by means of user-defined reductions.

Figure 2.1 shows the time-line of the OpenMP releases from the first Fortran version in 1997, until the upcoming OpenMP 5, by the end of 2018.

An important feature of OpenMP is the fact that neither the compiler nor the runtime must validate the conformity[1] of programs. The correctness of the code depends only on the programmer. Thus, frameworks do not need to check for issues such as data dependences, race conditions or deadlocks. As a result, the implementation of the standard is quite easy and light, and that boosts the spreading of the language even in architectures with few resources.

2.1.1.1 The execution model

OpenMP implements a fork-join model of parallelism. The program begins as a single thread of execution, called the *initial thread*, and parallelism is created through the `parallel` construct. When such a construct is found, a team of threads is spawned, which are joined at the implicit barrier encountered at the end of the parallel region. Within that region, the threads of the team execute work following two different patterns:

- The *thread-centric* model, which defines a conceptual abstraction of user-level threads that work as proxies for physical processors, enforcing a rather structured parallelism. Representative constructs are `for` and `sections`.

[1] An OpenMP program is *conforming* if it follows all rules defined in the specification.

- The *task-centric* model, which is oblivious of the physical layout, and allows programmers to focus on exposing parallelism rather than mapping parallelism onto threads. Representative constructs are `task` and `taskloop`. Furthermore, tasks can be either *tied*, if they are tied to the thread that starts the execution of the task, or *untied*, if they are not tied to any thread.

In OpenMP, mutual exclusion is accomplished via the `critical` and `atomic` constructs, and synchronization by means of the `barrier` construct. Additionally, the tasking model offers the `taskwait` construct to impose a less restrictive synchronization (while a barrier synchronizes all threads in the current team, a taskwait only synchronizes child tasks of the binding task[2], and the dependence clauses, that allow a data-flow driven synchronization among tasks). The values allowed for the dependence clauses are the following:

- `in`: a task with an l-value as input dependence is eligible to run when all previous tasks with the same l-value as output dependence have finished its execution.
- `out`: a task with an l-value as output dependence is eligible to run when all previous tasks with the same l-value as input or output dependence have finished its execution.
- `inout`: a with an l-value as inout dependence behaves as if it was an output dependence.

As an illustration, Listing 2.1 and 2.2 show the addition of two arrays using the `for` construct (thread model) and the `taskloop` construct (tasking model). In both cases, when the `parallel` construct is found, a team of threads is created. Also, in both cases all threads wait for the completion of the whole computation in the implicit barrier at the end of the parallel region. After that, all the threads are released and the *master thread*[3] keeps executing sequentially. The difference appears when the work has to be distributed. In the thread-centric example, the `for` worksharing splits the iteration space and distributes work among the different threads of the team, which will execute until there is no more work to do. On the other hand, in the task-centric example, a `single` construct is needed to indicate that just the master thread of the team will execute the code inside the region. Then, when the `taskloop` construct is found, a number of tasks is created and the iterations are distributed among them. These tasks can be executed immediately by one of the threads of the team, or may be deferred.

```
1 void add_arrays(int n, float *a,
2                 float *b, float* c)
3 {
4     #pragma omp parallel for
5     for (int i = 1; i < n; i++)
6         c[i] = a[i] + b[i];
7 }
```

Listing 2.1: Addition of two arrays using the OpenMP thread-centric model.

```
1 void add_arrays(int n, float *a,
2                 float *b, float* c)
3 {
4     #pragma omp parallel
5     #pragma omp master
6     #pragma omp taskloop
7     for (int i = 1; i < n; i++)
8         c[i] = a[i] + b[i];
9 }
```

Listing 2.2: Addition of two arrays using the OpenMP task-centric model.

[2]The *binding region* is the enclosing region that determines the execution context, and limits the scope of the effects of the bound region.

[3]The OpenMP specification defines the *master thread* as a thread that has thread number 0. This can be the *initial thread* or the thread that encounters a parallel construct, creates a team, generates a set of implicit tasks, and then executes one of those tasks as thread number 0.

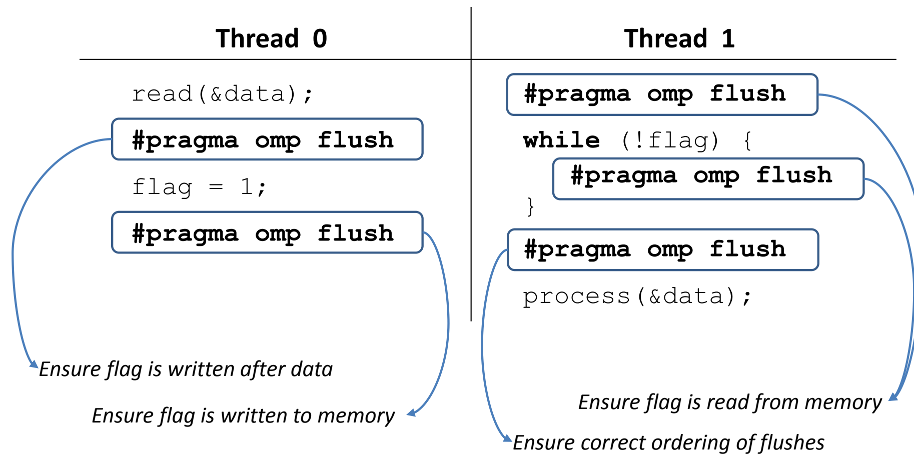


Figure 2.2: Producer-consumer pattern implemented using OpenMP flush constructs.

2.1.1.2 The memory model

OpenMP is based on a relaxed-consistency, shared-memory model. This means there is a memory space shared for all threads, called *memory*. Additionally, each thread has a temporary view of the memory. Intuitively, the temporary view is not always required to be consistent with the memory. Instead, each private view synchronizes with the main memory by means of the *flush* operation. Hence, memory operations can be freely reordered except around flushes. This synchronization can be implicit (in any, implicit or explicit, synchronization operation causing a memory fence) or explicit (using the `flush` directive). Data cannot be directly synchronized between two different threads temporary view. Figure 2.2 shows an example of a producer-consumer pattern implemented using OpenMP flushes. There, some flushes are used to ensure the memory consistency across the different views of the memory, and some others are used to ensure the correct order of execution, as explained in the figure.

The view each thread has for a given variable is defined using data-sharing clauses, which can determine the following sharing scopes:

– `private`: a new fresh variable is created within the scope.
– `firstprivate`: a new variable is created in the scope and initialized with the value of the original variable.
– `lastprivate`: a new variable is created within the scope and the original variable is updated at the end of the execution of the region.
– `shared`: the original variable is used in the scope, opening the possibility of race conditions.

The data-sharing attributes for variables referenced in a construct can be *predetermined*, *explicitly determined* or *implicitly determined*. Predetermined variables are those that, regardless of their occurrences, have a data-sharing attribute determined by the OpenMP model. Explicitly determined variables are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct. Implicitly determined variables are those that are referenced in a given construct, do not have predetermined data-sharing attributes and are not listed in a data-sharing attribute clause on the construct.

2.1.2 OmpSs

OmpSs [20, 45] is a parallel programming model developed at Barcelona Supercomputing Center (BSC) with the goal of working as a precursor of OpenMP in two main directions: asynchronous parallelism and heterogeneity. The former is accomplished be means of data-dependence clauses, and the latter through the `target` construct. The implementation of OmpSs is based on two tools: the Mercurium compiler (see Section 2.2.1 for further details), which transforms high-level directives into parallel code, and the Nanos++ [149] runtime system, which provides the services to manage parallelism. Although OpenMP and OmpSs have many similarities, this section introduces some important differences that concern both the execution and the memory models.

2.1.2.1 The execution model

In contrast to the fork-join model defined in OpenMP, parallelism is implicitly created when an OmpSs application starts. This means that the OmpSs model starts with a team of threads, the *initial team*, where there is a single *master thread* and a set of *worker threads*. The master thread is the one that starts running sequentially the user program. The rest of threads wait until concurrency is exposed (e.g., through a `task` or `for` directive). Instead, OpenMP starts the execution with a team of just one thread and creates and destroys a team of threads each time a `parallel` directive is found. Because the OmpSs model ignores any `parallel` construct, undesired results may appear if executing an OpenMP program in an OmpSs environment due to differences in the data accessibility within parallel regions.

Furthermore, OmpSs allows the annotation of function declarations or definitions in addition to structured-blocks. Thus, a function annotated with the `task` construct causes each invocation to become a task creation point. The evaluation of the arguments does not form part of the task. Since tasks may be deferred, a restriction forces such a task not to have any return value.

Finally, OmpSs offers a rich variety of mechanisms for fine-grain synchronization basides dependence clauses associated to OpenMP tasks, and the `taskwait` construct:

- new dependence clauses that can be associated to the `task` construct:
  * `concurrent`, a special version of the `inout` clause where the dependences are computed with respect to `in`, `out`, `inout` and `commutative`, but not to other `concurrent` clauses.
  * `commutative`, which forces to check dependences with respect to `in`, `out`, `inout` and `concurrent` clauses, but not other `commutative` clauses. Although different ready commutative tasks cannot run in parallel, they can run in any order.
- support for dependence clauses (`in`, `out` and `inout`) on the `taskwait` construct. This forces to synchronize only those tasks that declare dependences on the same variables the `taskwait` does.
- *multi-dependences*, a novel syntax to define a dynamic number of task dependences over an specific l-value. For a C/C++ program, the syntax, illustrated in Listing 2.3, is the following:

$$dependence - type(memory - reference - list, iterator - name = lower; size)$$

```
1 void foo(int n, int *v)
2 {
3    // This dependence is equivalent to inout(v[0], v[1], ..., v[n-1])
4    #pragma omp task inout({v[i], i=0;n})
5    {
6       int j;
7       for (int j = 0; j < n; ++j)
8          v[j]++;
9    }
10 }
```

Listing 2.3: OmpSs multidependences syntax example.

– Dependence clauses allow extended l-values from those of C/C++. Along with the array sections, already supported in OpenMP, OmpSs accepts *shaping expressions*, which allow recasting pointers into array types to recover the size of dimensions that could have been lost across function calls. A shaping expression is one or more *[size]* expressions before a pointer.

2.1.2.2 The memory model

OmpSs offers a single address space view, meaning that heterogeneous systems such as clusters of Symmetric Multi-Processing (SMP) machines and accelerators can be accessed as if only one memory address space existed. This feature allows OmpSs programs to run in different system configurations without being modified. For this property to hold, users are constrained to specify the data each task accesses using the following data-copying clauses:

– `copy_in(list-of-variables)` indicates the referenced shared data may need to be transfered to the device before the code associated to the task can be executed.
– `copy_out(list-of-variables)` indicates the referenced shared data may need to be transfered from the device after the code associated to the task is executed.
– `copy_inout(list-of-variables)` is equivalent to having `copy_in` and `copy_out` clauses for the same variable.
– `copy_deps(list-of-variables)` indicates to use the data-dependence clauses as if they were data-copying clauses.

2.1.3 Ada

Ada is a standard programming language where reliability and efficiency are essential. For that reason, it is specially focused on embedded systems, and it is widespread in high-integrity, safety-critical and high-security domains including commercial and military aircraft avionics, air traffic control, railroad systems, and medical devices. The whole language is designed to maintain safeness: it enforces strong typing, checks ranges in loops and so eliminates buffer overflows, provides actual contracts in the form of pre- and post-conditions, prevents access to deallocated memory, etc. A long list of language decisions allows compilers to implement correctness techniques to certify algorithms regarding their specification.

2.1.3.1 Concurrency model

Ada includes tasking features as part of the language standard. A *task* is a language entity of concurrent execution, with its internal state and defined behavior. There is a conceptual task, called the *environment task*, which is responsible for the program elaboration. This task is generally the operating system thread which initializes the runtime and executes the main subprogram. Before calling the main procedure, the environment task elaborates all library units referenced to in the main procedure. This elaboration will cause library-level tasks to be created and activated before the main procedure is called. There are two types of tasks:

- *Declared tasks*: these start (are activated) implicitly when the parent unit begins.
- *Dynamic tasks*: these start (are activated) at the point of allocation. This reduces the possibility of a high start-up overhead.

A task (both declared and dynamic) completes execution by reaching the end of the task body. A local task (that is, a task declared within a subprogram, block, or another task) must finish before the enclosing unit can itself be left, so the enclosing unit will be suspended until the local task terminates. This rule prevents dangling references[4] to data that no longer exist. Additionally, tasks can be created as a one of a kind *task* or as a *task type*, which can be used to create many identical task objects. Both are defined in two parts: the first part defines the public interface of the task, and the second part contains the implementation of the task code (body).

Figure 2.3 shows a simplified diagram of the states of a task and the transitions among them. A task is *inactive* when it has just been created. After the runtime associates a thread of control to this task, it can be *terminated*, if the elaboration of the task fails, or *runnable* instead, so the user code of the task is executed. If this code executes some operation that blocks the task (protected operation, *rendezvous* or `delay` statement) it goes to *sleep* and later returns to the *runnable* state. When the task executes a `terminate` alternative or finalizes the execution of the user code, it is *terminated*. The complete diagram of task states and transitions is shown in Appendix A.

The mapping of the tasks to processors can be handled by the runtime/operating system [107] or, since Ada 2012, statically assigned by the programmer [1].

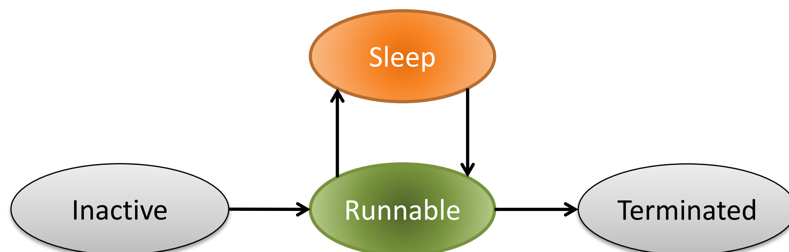


Figure 2.3: Diagram of Ada task states.

[4] A *dangling reference* is a reference (address) that does not resolve to a valid destination. This may happen when accessing an object through a pointer after the object has been freed: either a local variable that has gone out of scope, or a dynamically allocated object that has been explicitly freed through some other pointer.

2.1.3.1.1 Mutual exclusion

Mutually exclusive access to shared data is a necessary feature for concurrent languages to avoid race conditions. Ada offers two different mechanisms to achieve mutual exclusion: protected objects (with an associated locking policy), further divided into non-blocking and blocking operations, and the *rendezvous*.

Protected procedures and protected functions Protected procedures and protected functions are non-blocking operations. They differ in that protected procedures provide mutually exclusive read/write access to the encapsulated data, and protected functions provide concurrent read-only access to the encapsulated data. Hence, several function calls can run simultaneously, but calls to a protected function are still executed mutually exclusive with calls to a protected procedure. Communication between tasks using protected objects for data sharing is asynchronous.

Protected entries Protected entries are similar to protected procedures in that they offer mutually exclusive read/write access to the protected data, and differ in that protected entries are guarded by a boolean expression. When the boolean evaluates to false, the calling task is suspended until the condition evaluates to true and no other task is active inside the protected object. Each entry has an associated queue where the callers that has been suspended are stored. Then, when a subprogram with a write access to a protected object finishes, all conditions with queued tasks are re-evaluated. The core language does not specify any particular order in the execution of the tasks in an entry's queue, however the real-time annex (annex D [3]) provides specific rules in the *Priority_Queueing* policy [5][5]. Ada also specifies language mechanisms for the calling task to timeout (*timed* [2]), or canceling a call if the condition is not true (*conditional entry calls* [4]). Protected entries are a conditional synchronization mechanism.

Rendezvous The *rendezvous* mechanism is based on a *client/server model*. Clients are active objects that initiate spontaneous actions, and servers are reactive objects that perform actions only when invoked by active objects. Thus, a server task offers services to the client tasks by declaring public entries in its specification, and rendezvous is requested when a task calls an entry of another task. For the rendezvous to take place, the called task must *accept* the entry call. Meanwhile, the calling task waits while the accepting task executes and, when the accepting task ends the rendezvous, both tasks may continue their execution. If a client task calls an entry of a server task, and this is not waiting at an `accept` statement for that entry, then the caller is queued. Alternatively, if the server task reaches an `accept` and no task is waiting on the associated queue, then the server is suspended. Server tasks may use the *selective wait* statement to allow a task to wait for a call on any entry, avoiding to be held-up on a particular entry. The rendezvous synchronous behavior is hard to analyze in the context of the real-time domain due to two main reasons: a) the time expended in the waiting queues cannot be evaluated, and b) the non-deterministic selection of entry calls introduces unpredictability. For these reasons, this model of communication is not used for real-time applications.

[5] The variable and unknown arrival instants of calls to a protected entry introduce non-determinism. For this reason, a sound Ada program shall not depend on a particular order of execution of pending entry calls.

Examples Listings 2.4, 2.5 and 2.6 show different approaches to implement a Stock data structure. The first illustrates non-blocking operations in the form of procedures. In this case, when the procedures complete, the corresponding output variable, *Full* or *Empty*, is updated accordingly. The second illustrates equivalent behavior with blocking operations in the form of entries. In this case, the calling tasks block when the condition (*Full* or *Empty*) is true. And the third shows a server task that implements synchronous communication for updating the *Stock*.

```
1  protected Stock is
2    procedure Add(N : in Integer; Full: out Boolean);
3    procedure Remove(N : in Integer; Empty: out Boolean);
4  private
5    Total : Integer := 0;
6  end Stock;
7  protected body Stock is begin
8    procedure Add(N : in Integer; Full: out Boolean) is begin
9      if N + Total > 1000 then
10       Total := 1000; Full := True;
11     else
12       Total := Total + N; Full := False;
13     end if;
14   end Add;
15   procedure Remove(N : in Integer; Empty: out Boolean) is begin
16     if N > Total then
17       Total := 0; Empty := True;
18     else
19       Total := Total - N; Empty := False;
20     end if;
21   end Remove;
22 end Stock;
```

Listing 2.4: Stock data-structure implemented using synchronization-free Ada protected objects.

```
1  protected Stock is
2    entry Add(N : in Integer);
3    entry Remove(N : in Integer);
4  private
5    Total : Integer := 0;
6  end Stock;
7  protected body Stock is begin
8    entry Add(N : in Integer)
9              when Total < 1000 is
10   begin
11     if N + Total > 1000 then
12       Total := 1000;
13     else
14       Total := Total + N;
15     end if;
16   end Add;
17   entry Remove(N : in Integer)
18                 when Total > 0 is
19   begin
20     if N > Total then
21       Total := 0;
22     else
23       Total := Total - N;
24     end if;
25   end Remove;
26 end Stock;
```

Listing 2.5: Stock data-structure implemented using Ada protected objects with synchronization.

```
1  task Stock is
2    entry Add(N : in Integer);
3    entry Remove(N : in Integer);
4  end Stock;
5  task body Stock is
6    Total : Integer := 0;
7  begin
8    loop
9      select when Total < 1000 =>
10         accept Add (N : in Integer) do
11           if N + Total > 1000 then
12             Total := 1000;
13           else
14             Total := Total + N;
15           end if;
16         end Add;
17     or when Total > 0
18         accept Remove (N : in Integer) do
19           if N > Total then
20             Total := 0;
21           else
22             Total := Total - N;
23           end if;
24         end Remove;
25     end select;
26   end loop;
27 end Stock;
```

Listing 2.6: Stock data-structure implemented using the Ada rendezvous mechanism.

2.1.3.2 Safety

One key aspect in the Ada is that the language preserves the system's integrity, considering safety (the software must not harm the world) and security (the world must not harm the software) [23]. For that reason, Ada introduces restrictions and checks that allow certifying software with respect to its specification. Among the most important considerations, Ada provides: a robust syntax that prevents typing errors, strong typing, access types modeling pointers that avoid the most common errors associated to this objects (type safety violations, dangling references and storage exhaustion), checks for avoiding memory leaks such as buffer overflow, and mechanisms for optimizing the use of the stack and the heap (storage pools).

Additionally, the language introduces the type Time, which is used in `delay` statements timed entry calls, and timed selective waits. The existence of these features makes Ada very convenient for real-time systems, where time constraints must be controlled as part of the safety requirements.

Furthermore, the concurrency model of Ada has been designed to fulfill the safety requirements of the language. The fact that Ada provides tasking facilities within the language by means of built-in syntactic constructions has two main advantages: a) Ada provides a level of abstraction that hides low level details and thus prevents certain errors from being made, and b) built-in constructions allow the compiler to be aware of concurrent execution and thus detect some data races. As a result, the typically used operations in a tasking program are safe:

- tasks can be prevented from violating the integrity of data.
- tasks can be controlled in order to meet specific timing requirements.
- tasks can be scheduled in order to use resources efficiently and to meet their overall deadlines.

The Ravenscar profile In 1997 appeared the Ravenscar profile [34], a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness (note that the profile does not address any non-tasking aspect of the language[6]). The rationale behind this restriction of the language is summarized as follows:

- The set of tasks and interrupts must be fixed and have static properties after program elaboration, so it can be analyzed to prove its safety (no task should get into an unsafe state such as a deadlock or a livelock) and liveness (all desirable states of the task must be reached eventually). Related restrictions include preventing dynamic priorities (*No_Dynamic_Priorities*), abort statements (*No_Abort_Statements*), task hierarchies (*No_Task_Hierarchy*), and task termination (*No_Task_Termination*).
- The set of protected objects must be known statically to allow schedulability analysis, and these cannot be declared locally, because then they are meaningless for mutual exclusion and task synchronization. Related restrictions include preventing local protected objects (*No_Local_Protected_Objects*), select statements (*No_Select_Statements*), and task entries avoiding the rendezvous mechanism (*Max_Task_Entries => 0*).
- Memory cannot be dynamically allocated. Related restrictions are preventing from implicit head allocations (*No_Implicit_Heap_Allocations*), and the use of the

[6]The sequential aspects of Ada (such as exception handling) are covered in the "Guide for the Use of the Ada Programming Language in High Integrity Systems" [162], where different forms of static analysis are proposed.

Ada.Task_Attributes package (`No_Task_Attributes_Package`), which may allocate memory dynamically to store task attributes.

- Execution must be deterministic. Related restrictions include limiting the number of protected entries to 1 entry (`Max_Protected_Entries => 1`), limiting the length of the queues to 1 element (`Max_Entry_Queue_Length => 1`), and the use of a high precision timing mechanism (restriction `No_Calendar` forces use of the Ada.Real_Time time type, or an implementation defined time type).

Furthermore, the Ravenscar profile defines a series of dynamic semantics that include:

- The required *task dispatching policy* is FIFO_Within_Priorities, which is a FIFO queue where the *active priority*[7] of the tasks is taken into account.
- The required *locking policy*[8] is Ceiling_Locking, because it provides one of the lowest worst case blocking times for contention for shared resources, hence maximizing schedulability when preemptive scheduling is used. Using this policy, a task executing a protected action inherits the ceiling priority of the corresponding protected object during that execution.
- The *queuing policy* is meaningless in the Ravenscar profile because no entry queues can form.
- The profile drastically reduces the number of *runtime errors* to two: a) violation of the priority ceiling, and b) more that one task waiting concurrently on a suspension object. On the other hand, the profile also introduces some additional two concurrency-related checks: a) checking that all task are non-terminating, and b) checking that the maximum number of calls that are queued concurrently on an entry does not exceed one.

2.2 Execution environment

The execution environment related to this thesis mainly involves two tools: the Mercurium compiler and the libgomp runtime library. The following sections introduce both systems.

2.2.1 The Mercurium source-to-source compiler

Mercurium [52] is a source-to-source compiler developed by the Programming Models [22] group at BSC. Dedicated to research, its main goal is to provide an infrastructure for fast prototyping of new parallel programming models. It currently has support for C99 [128], C++11 [73] and Fortran 95 [118] languages, and also implements several programming models such as OpenMP 3.1 [110], OmpSs [15], CUDA [40] and OpenCL [143].

Mercurium uses a plugin architecture, where each plugin represents a phase of the compiler. Figure 2.4 shows a high-level scheme with the most representative phases of the compiler relevant to this thesis. The compiler is structured in three main parts. The first one is a front-end that fully supports C, C++ and Fortran. During this part of the compilation, the system gathers symbolic and typing information, and structures it in an Intermediate Representation (IR) shared by all

[7]The *priority* of an Ada task is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. The *base priority* of a task is the priority with which it was created, or to which it was later set by Dynamic_Priorities.Set_Priority. At all times, a task also has an *active priority*, which generally reflects its *base priority* as well as any priority it inherits from other sources [6].

[8]The Ada *locking policy* specifies the interactions between priority task scheduling and protected object ceilings.

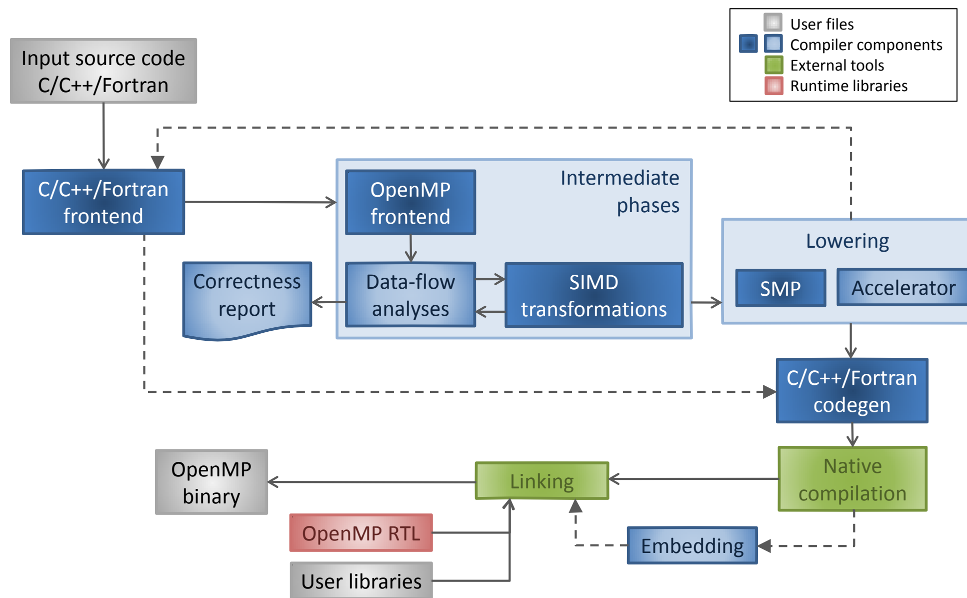


Figure 2.4: Mercurium compilation diagram.

three languages. The second one is a pipelined sequence of phases that performs source-to-source transformations to meet the target programming model and runtime. Finally, the last part is the code generator. During this part, the compiler generates the final source code based on the IR. The generated output is compiled and linked with the corresponding native compiler.

2.2.1.1 Intermediate representation

Mercurium's front-end parses the source code to generate an Abstract Syntax Tree (AST)[9] that holds an accurate high-level representation of the input. Classical type checking is performed using the AST to create a symbol table for each scope and remove ambiguities, while synthesizing expression types. The result of this step is a non-ambiguous tree, called *Nodecl*, which is the IR that will be used in the different phases of the compiler. The IR is also an AST, but differs from the initial one in some aspects: a) it does not contain declarations, instead includes *context* nodes everywhere a block of code creates a new scope, and b) it represents with the same structure all C, C++ and Fortran languages, easing the following phases of the compiler since they will be (almost) language-independent. Symbolic data such as types, symbols and scopes (i.e., global, namespace, function block and current) are stored separately from the AST, although they are accessible from the corresponding nodes. Figure 2.5 shows a simplified version of the AST generated for the OpenMP code computing the Fibonacci sequence depicted in Listing 2.7. Nodes of the AST are printed in black, whereas information about symbols is printed in orange, information about types in green, and information about scopes in blue (for ease of reading, we illustrate the scoping information only of two context nodes). Additionally, nodes holding parallel semantics have gray background.

[9]An *abstract syntax tree* is a tree representation of the abstract syntactic structure of a source code written in a programming language.

```
1  int fib(int n) {
2      int i, j;
3      if (n < 2)
4          return n;
5
6      #pragma omp task shared(i)
7          i = fib(n - 1);
8      #pragma omp task shared(j)
9          j = fib(n - 2);
10     #pragma omp taskwait
11     return i + j;
12 }
```

Listing 2.7: Recursive Fibonacci computation using OpenMP tasks.

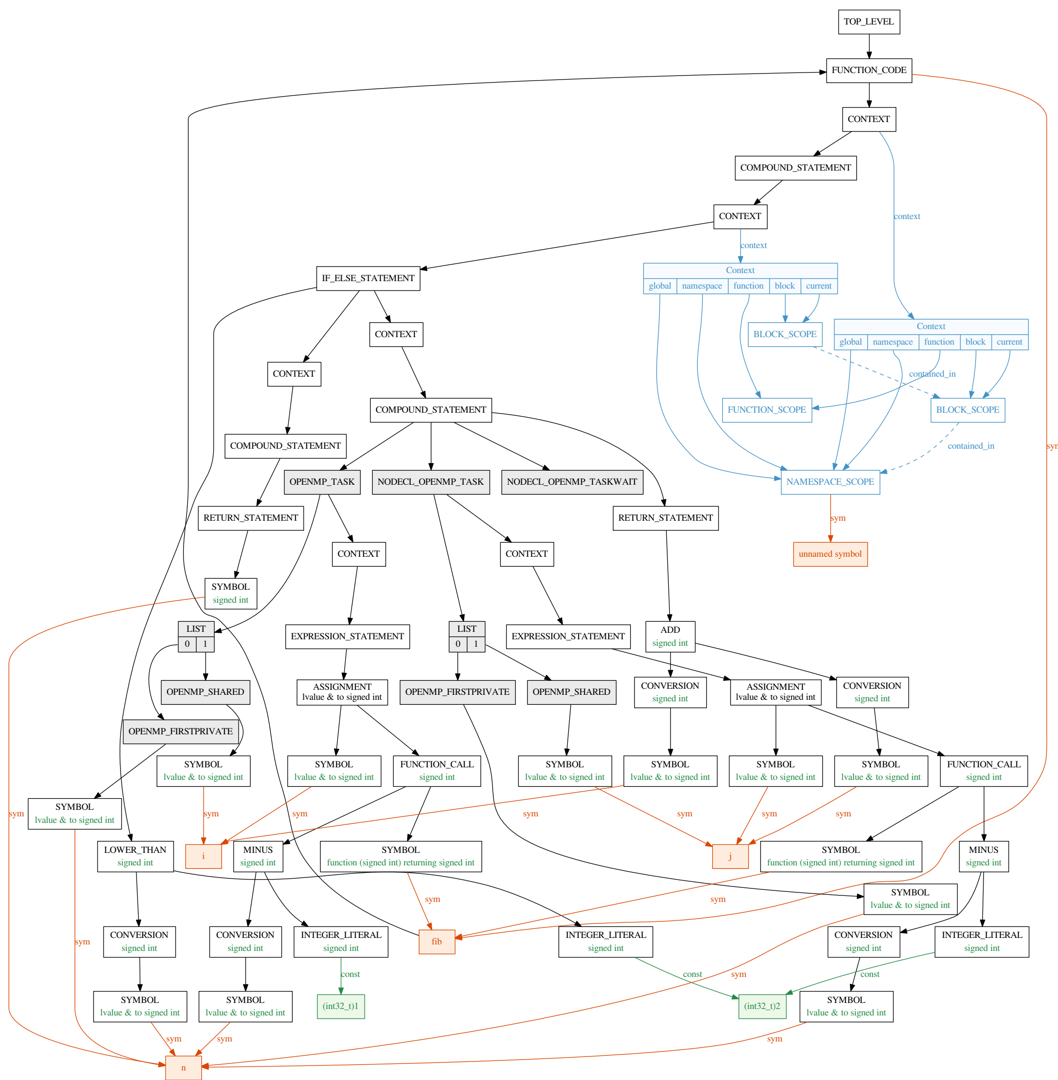


Figure 2.5: Mercurium simplified AST for Fibonacci computation in Listing 2.7.

A single grammar describes the AST nodes that represent all C, C++ and Fortran. Most of the nodes are shared by the three languages, nonetheless there are a few specific nodes aimed at representing their particularities. This grammar also describes OpenMP and OmpSs semantics with specific nodes. As an illustration, Listing 2.8 shows different (incomplete) rules: OpenMP (*parallel-execution* and *task-construct* rules), expressions shared among all languages (*add* and *minus* nodes in *expression* rule), expressions specific to C/C++ (*predecrement* and *throw* nodes in *c-cxx-expressions* rule), and expressions specific to Fortran (*boz literal* and *use only* nodes in *fortran-expressions* rule). Each AST node is composed of: 1) a kind, 2) up to 4 child nodes, and 3) a set of external attributes that are node-kind dependent (e.g., the type of the node, a symbol, a scope, and an associated constant value). This grammatical description is translated to a non-hierarchical class system that is then used in the subsequent phases of the compiler.

```
1  parallel-execution : task-construct
2                     | parallel-construct
3                     | critical-construct
4                     | ...
5
6  task-construct : NODECL_OPEN_M_P*TASK([environment]omp-exec-environment-seq-opt,
7                                        [statements]statement-seq-opt)
8
9  expression : NODECL_ADD([lhs]expression, [rhs]expression) type const-value-opt
10            | NODECL_MINUS([lhs]expression, [rhs]expression) type const-value-opt
11            | c-cxx-expressions
12            | fortran-expressions
13            | ...
14
15 c-cxx-expressions : NODECL_PREDECREMENT([rhs]expression) type const-value-opt
16                   | NODECL_THROW([rhs]expression-opt) type const-value-opt
17                   | ...
18
19 fortran-expressions : NODECL_FORTRAN_BOZ_LITERAL() type text const-value
20                     | NODECL_FORTRAN_USE_ONLY([module]name, [only_items]name-seq)
21                     | ...
```

Listing 2.8: Example of rules of the grammar generating the Mercurium's IR.

2.2.1.2 Compiler phases

Mercurium implements a set of phases that translate the code modeled in the high-level IR generated by the front-end. These phases transform the code guided by user directives. The compiler accepts a relaxed form of the *pragma directive* syntax, meaning that the initial parsing only recognizes tokens (directive name, clauses, and expressions and values associated with these clauses). It is a subsequent phase that gives meaning to the different directives and clauses while it checks them for correctness. The compiler front-end and initial semantic passes offer the basic IR to the rest of the passes. Each transformation phase can enrich the IR with new information, which can be later used by subsequent passes.

Each compiler pass is implemented as a shared library, and is dynamically loaded into the compiler process as a plugin. This flow is driven by a configuration file associated with the driver that is used to invoke the compilation. The configuration file describes the way that a particular invocation of the compiler must proceed. Listing 2.9 shows a portion of the configuration file used

```
1 # when --instrument is given, activate an internal variable indicating so
2 {instrument} options = --variable=instr:1
3 # load the proper compiler plugin
4 {instrument} compiler_phase = libtlinstr.so
5 # and link against the proper (instrumented) libraries
6 {instrument} linker_options = \-L@NANOX_LIBS@/instrumentation -lnanox
```

Listing 2.9: Example of a configuration file of Mercurium.

for instrumenting the code, where a set of compiler passes are indicated to be loaded and executed in order. This passes can be conditionally activated given a compiler flag, as it happens with the *instrument* passes.

The Data Transfer Object (DTO) pattern is used to transfer data between phases. The DTO is just a dictionary containing a string as the key, and an object as the value. At any point of the compilation process we can find available the *translation unit*[10] IR with the processed code. The visitor pattern is implemented to perform traversals through the *Nodecl*, this way the traversal is completely separated from the operation to be performed.

2.2.2 The *libgomp* runtime library

The GOMP project [60] has implemented support for OpenMP in the GNU Compiler Collection by means of two tools: a) the OpenMP compiler, GOMP, which is an extension of the GCC compiler that converts OpenMP directives into threading calls, and b) the OpenMP runtime library, *libgomp*, which allows parallel execution by mapping OpenMP threads onto different thread implementations. Although current versions of GCC (6 and later) support OpenMP 4.5, we have used GCC 5.4 [56], which implements support for OpenMP 4.0 plus offloading [58], because it was the latest version at the moment this part of the project was developed. In the context of this thesis, we are interested only in those aspects regarding the definition and scheduling of OpenMP tasks. The following paragraphs introduce the operation of GOMP and libgomp for those matters.

In regard to the compiler, when a task construct is found, GCC offloads the code within the directive into a function, wraps the arguments to be passed to the outlined function in a data structure, and replaces the whole directive with a call to *GOMP_task* (the libgomp function that implements the creation of a task). This function receives different arguments: a pointer to the outlined function, a pointer to the data structure with the arguments of the outlined function, and information about the dependences, among others. Specifically, dependences are stored as a `void**` array containing the addresses of the variables referenced in the dependence clauses.

In regard to the runtime, when the method *GOMP_task* is called, a new task is created. This task can either be executed immediately (e.g., the task is created inside a *final* task, or the task has an `if` clause that evaluates to false), or deferred (e.g., the task has unfulfilled dependences). The library uses three different queues to store the tasks that are ready to be executed (either because they do not have dependences, or because their dependences have been fulfilled): 1) the

[10] According to the standard C++[73], a *translation unit* is the basic unit of compilation in C++, which consists of the contents of a single source file, plus the contents of any header files directly or indirectly included by it, minus those lines that were ignored using conditional preprocessing statements.

ready tasks queue of the team to which the task belongs (this queue is used when a barrier is encountered); 2) the *children queue* of the parent of the task (this queue is used when a taskwait is encountered); and 3) the children queue of the taskgroup to which the task belongs (if the task is inside a taskgroup directive). Additionally, the library uses a hash table per task region to manage the tasks with unfulfilled dependences. The hash table is addressed with the address of each dependence variable, and each key points to the list of tasks that define that dependence. Furthermore, each task also stores pointers to its dependent tasks.

According to the previous explanation, when a new task is created, the variables in its dependence clauses are checked in the hash table. If none is found, the task is included in the proper queues as it is ready to be executed. Otherwise, it is stored in the hash table of its parent task. A counter keeps track of all the unfulfilled dependences of a task. When this counter reaches zero, the task is added to the queues to be instantiated when possible.

2.3 Architectures

Different parts of this thesis have been evaluated on two kind of architectures: SMP machines and embedded multi-processors. This section introduces the most important features of these architectures, as well as the details of the specific machines that have been used.

2.3.1 HPC architectures: Intel Xeon

21st century supercomputers can use over 100,000 processors, combining CPUs with GPUs, connected by fast connections. Typically, processors are first combined in an on-board SMP system. Then, several such systems are likewise combined to form a Non-Unified Memory Access (NUMA) system. Different NUMA systems are in turn integrated into clusters, which finally communicate on a grid organization.

SMP involves a multiprocessor architecture where several symmetric processors (all processors can perform the same functions) are connected to a single, shared memory. For that reason, all processors require similar time to access any part of the memory. Additionally, all processors have full access to all I/O devices, and are controlled by a single operating system instance. All the components (memory, I/O devices, processors, etc.) are connected using a system bus, a crossbar switch or a mesh topology. Figure 2.6 depicts a high-level diagram of a common SMP system connected through a single bus.

NUMA is a type of shared memory architecture that involves several processors that can access their own local memory faster than non-local memory (the local memory of other processors or memory shared among processors). A particular version of this kind of architecture, cache-coherent NUMA systems (ccNUMA), maintains cache coherence across shared memory by using inter-processor communication between cache controllers. Many NUMA environments can be implemented as if they were SMP machines because the details of the architecture are hidden from the user. Figure 2.7 depicts a high-level diagram of a common NUMA system, connecting four SMP systems.

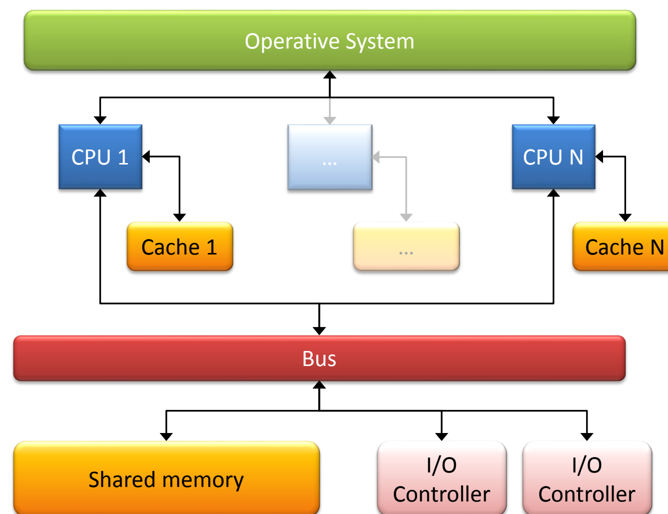


Figure 2.6: Diagram of an SMP system.

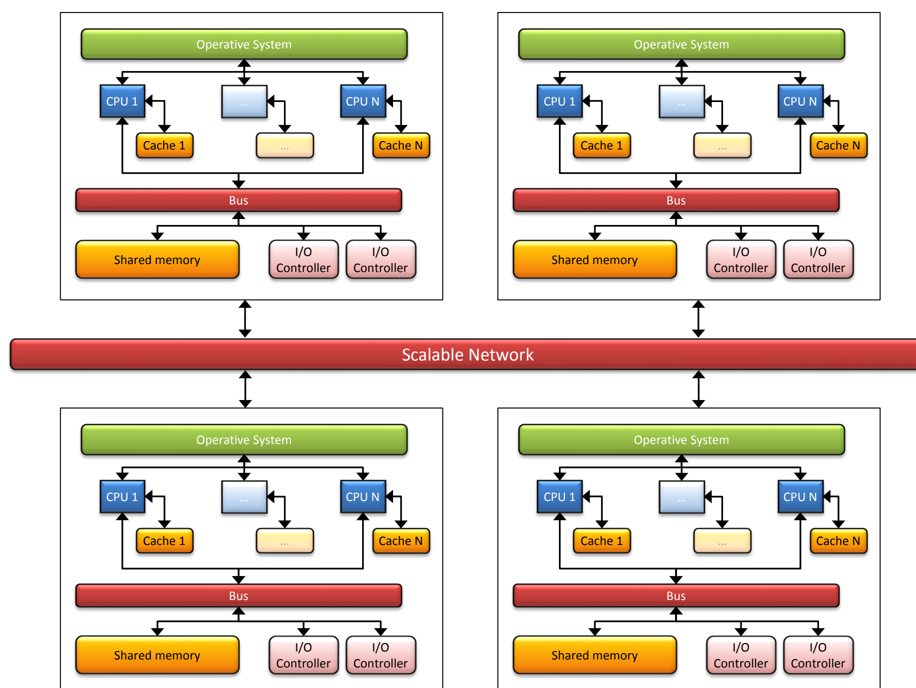


Figure 2.7: Diagram of a NUMA system.

A computer cluster consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system. Computer clusters have each node set to perform the same task, controlled and scheduled by software. The components of a cluster are usually connected to each other through fast local area networks, with each node running its own instance of the same operating system.

Finally, grid computing involves many networked loosely coupled computers acting together to perform large tasks. Unlike clusters, each node of the grid is set to perform a different task/application. Additionally, grid computers also tend to be more heterogeneous and geographically dispersed.

Different HPC architectures have been used in the context of this thesis. Those dedicated to evaluate correctness are not presented because the specific details are not relevant to the results. The HPC systems used for performance comparisons are Marenostrum III [18] and Marenostrum IV [19]. Both supercomputers are based on Intel Xeon processors. The former is composed by 37 iDataPlex compute racks, with 84 IBM dx360 M4 compute nodes each. Each node has two E5-2670 SandyBridge-EP processors (illustrated in Figure 2.8) running at 2.6GHz with a thermal design power of 115 Watts, and featuring 8 cores each with 20MB L3. The latter has a general purpose block composed by 48 SD530 compute racks housing 3456 compute nodes. Each node has two Intel Xeon Platinum 8160 processors (illustrated in Figure 2.9) running at 2.1GHz with a thermal design power of 150 Watts, and featuring 24 cores each with 33MB L3.

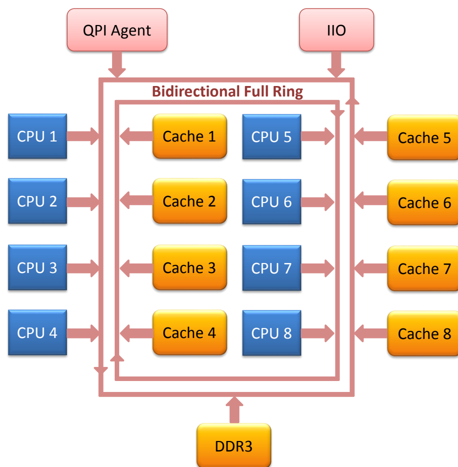


Figure 2.8: High-level view of the Intel Xeon E5-2670.

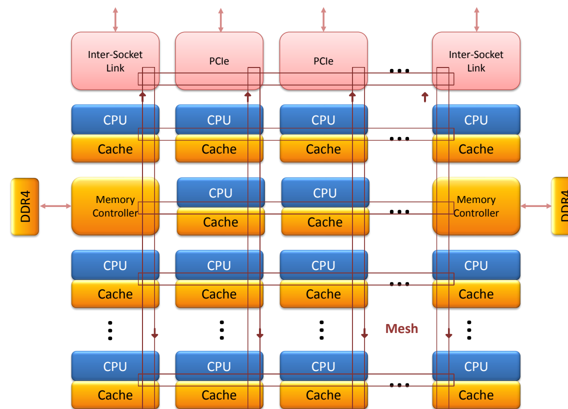


Figure 2.9: High-level view of the Intel Xeon Platinum 8160.

2.3.2 Real-time embedded architectures: the Kalray MPPA® processor

Multiprocessor System-on-Chip (MPSoC) systems are widespread (e.g., communications, vehicle systems and traffic control among many others), and their processing capacities are increasing. However, programmers not always make the most of these capacities because the development of parallel programs for those systems is difficult. Basically, the heterogeneity of the different embedded systems that currently exist prevent portability. Furthermore, the complexity of the systems forces programmers to manage low-level aspects such as scheduling work units and managing synchronizations between cores. Furthermore, many embedded systems are used for real-time purposes, which imposes even more constraints: guaranteed worst-case response times to critical events, and acceptable average-case response times to non-critical events.

Different proposals exist to adopt OpenMP as an standard for implementing parallel embedded architectures. However, support in the architecture side is still scarce. In that sense, the Kalray MPPA® processor has some advantages over other embedded multi-cores because it does support OpenMP. The MPPA® is a single-chip programmable 256-core processor manufactured in 28nm CMOS technology. The processor targets low to medium volume professional applications, where low energy per operation and time predictability are the primary requirements [91]. It has an operating frequency of 400MHz and a typical power consumption of 5 Watts, performing up to 700 Giga operations per second and 230 GFLOPS.

The MPPA® integrates a total of 288 identical Very Long Instruction Word (VLIW) cores including 256 user cores (processing engines, PE) dedicated to the execution of the user applications, and 32 system cores (resource manager, RM) dedicated to the management of the software and processing resources. Figure 2.10 shows a high-level description of this architecture. Cores are organized in 16 compute clusters (the inner blue boxes) holding 16 PEs and 1 RM each, and 4 I/O subsystems (the green boxes located at the periphery of the chip) holding 4 RMs each (the orange boxes). Each compute cluster and I/O subsystem owns a private address space with a total capacity of 2MB. The 4 IOS are dedicated to PCIe, Ethernet, Interlaken and other I/O devices, and each one runs a rich OS such as Linux or a RTOS that supports the MPPA I/O drivers. Communication and synchronization between different I/Os and clusters is ensured by data and control Networks-On-Chip (D-NoC and C-NoC, respectively). The former is optimized for bulk data transfers, while the latter is optimized for small messages at low latency. At this level, program parallelism is provided.

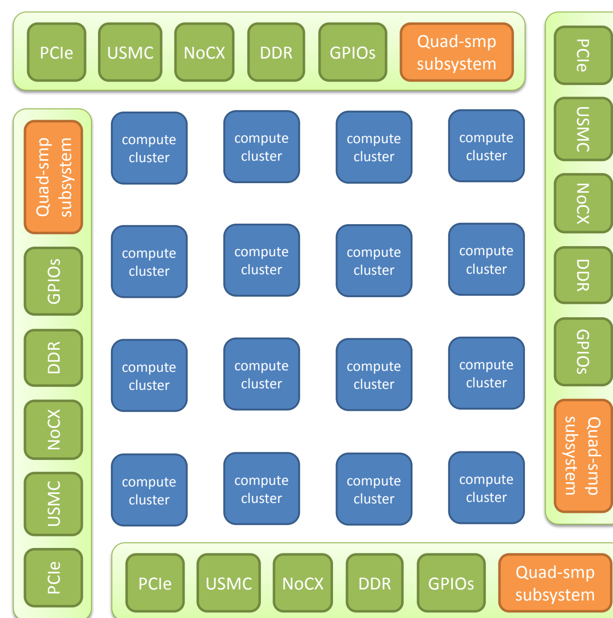


Figure 2.10: High-level view of the MPPA processor.

Figure 2.11 shows the high-level description of a compute cluster, where thread parallelism is exposed by means of a POSIX-based programming. This enables OpenMP, the implementation of which is based on a proprietary compiler and runtime for the K1 processor. Furthemore, Figure 2.12 shows the high-level description of a VLIW core, which exploits the instruction-level parallelism. This unit is able to execute up to five RISC-like instructions every cycle and eliminates timing anomalies[11] to support accurate static timing analysis [41].

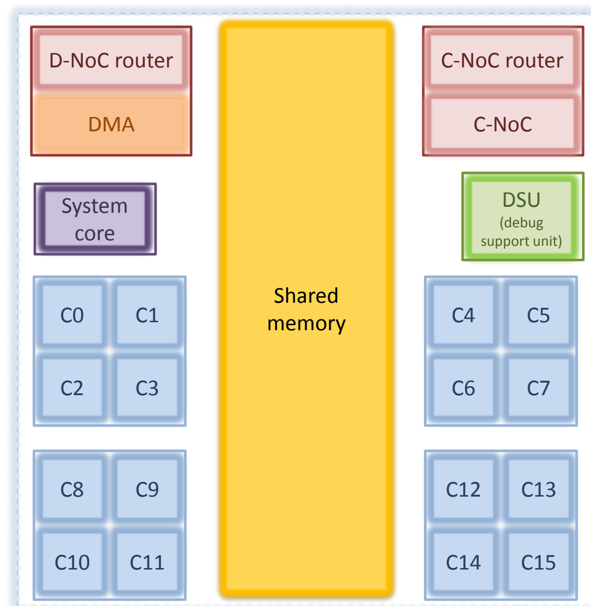


Figure 2.11: High-level view of an MPPA compute cluster.

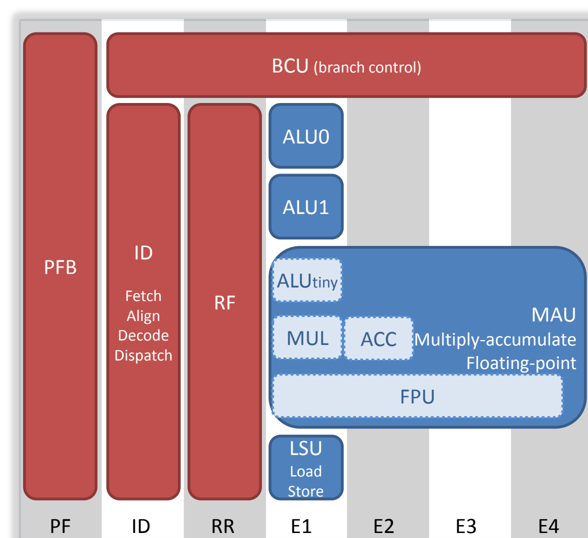


Figure 2.12: High-level view of an MPPA VLIW core.

[11] A timing anomaly is a situation where a local worst-case execution time does not contribute to the global worst-case.

3

Compiler analysis for OpenMP

The compilation process not only translates code from one language to another, but also performs a series of analyses with two main purposes: optimize the code and check its correctness. Classic compiler techniques include control-flow and data-flow analysis. The former aims to discover the hierarchical flow of control within a procedure, which is typically represented using a type of directed graph called Control Flow Graph (CFG). This representation allows the analysis of the different execution paths of a program. The latter, data-flow analysis, comprises several techniques aiming to gather information about the possible set of values calculated at various points of a program. Data-flow analyses are flow-sensitive and, generally, a CFG is used to traverse the structure of the program and perform calculations in each portion of code.

The incorporation of parallelism represents a challenge when it comes to efficiently represent the structure of the program and its parallel semantics. Furthermore, each parallel programming model represents its own paradigm, implementing different constructions and semantics. Focusing on OpenMP, the main challenges we faced regarding control-flow and data-flow analyses were two: the relaxed memory consistency model, and the unstructured parallelism of the tasking model (further details about OpenMP can be found in Section 2.1.1).

The following sections in this chapter explain the details of the analyses we extended to support OpenMP. All of them are implemented in the Mercurium compiler (see further details about the compiler in Section 2.2.1). Although Mercurium keeps a unique representation for all C, C++ and Fortran, we have focused on C and C++ codes. Nonetheless, we have performed some proofs of concept to extend the following analyses to Fortran, and the task has been quite straight-forward due to the common representation it uses to model all languages.

3.1 Internal Representation of the code

Traditionally, the Static Single Assignment (SSA)[1] form has been used for a wide range of compiler analysis and optimizations that include use-definition chains and reaching definitions among the former, and constant propagation and dead code elimination among the latter. This is so because it simplifies the properties of the variables, and hence facilitates the analysis.

[1]SSA is a property of an intermediate representation that requires that each variable is assigned exactly once, and also forces every variable to be defined before it is used.

However, SSA is a low level representation that transforms the code in such a way that, most of the times, it does not allow to return to the original code. This is not a suitable property in the context of this thesis because we intend to provide information to the user at a high level of abstraction. Furthermore, this thesis does not focus on code optimization, but on correctness, where the benefits of SSA are still to be proved. For these reasons, the analyses presented in the following sections are not based on SSA, but on the high-level representation of the Mercurium compiler, which allows us to communicate with the user in a more understandable way, and to reproduce the code almost as it originally was.

3.2 Classic analysis adapted to OpenMP

3.2.1 The Parallel Control Flow Graph

A CFG is a directed graph $\mathscr{G} = (N, E)$ where N is a set of nodes each representing a basic block[2], and E is a set of directed edges each connecting a pair of nodes. To describe the parallel model implemented in OpenMP, we draw from two Parallel Control Flow Graph (PCFG) representations developed at the same time as this work: the one developed in the OpenUH compiler [64, 65], and the one developed in the Cetus compiler [26]. These representations extend the classic CFG expressing parallelism based on barrier and flush synchronizations. Both descriptions are focused on the OpenMP thread-centric model of parallelism. We focus on the tasking model instead. As a result, our PCFG implements support for tasks synchronizations and, additionally, simplifies the structure by reducing the number of different types of both nodes and edges. As for classic CFGs, the scope of the constructed graph is a function. This allows both intra- and inter-procedural analysis. While intra-procedural analysis is self-contained in a function's graph, inter-procedural analysis is achieved by propagating the argument values of a function call to its corresponding PCFG. Besides, the return value of the called function is propagated back when possible.

A PCFG $\mathscr{G}$ is a tuple

$$\mathscr{G} = \langle N, n_{EN}, n_{EX}, E_F, E_C, E_S \rangle$$

where:

- N is the set of nodes. There are two types of nodes: *simple* nodes, representing either sequential execution of one or more statements, or parallel semantics for stand-alone directives (e.g., `barrier` or `flush`); and *structured* nodes, representing control flow (selection and iteration statements) or parallel semantics which have some user code associated (e.g., `task` or `parallel` constructs). *Structured* nodes are PCFGs themselves.
- $n_{EN} \in N$ and $n_{EX} \in N$ are the entry and exit nodes. These are the unique entry and exit points respectively, unless a `goto` statement jumps into or out of the graph.
- $E_F \subseteq N \times N \times L$ is the set of *flow edges* which correspond to the usual control flow of a program. Given an edge $(n_1, n_2, l) \in E_F$, n_1 is the *source*, n_2 is the *target*, and l is the label, where $L = \{\mathrm{u}, \mathrm{t}, \mathrm{f}\}$ (standing for *unconditional*, *true* and *false*, respectively).

[2] A *basic block* is a code sequence with no branches in except to the entry, and no branches out except at the exit.

- $E_C \subseteq N \times N$ is the set of *task creation edges*. When the program execution encounters a `task` construct, a task is created. Since the task may be deferred, there is no *flow edge* from the *task creation node* n_{TC} to the *task statement node* n_T (*task node* henceforth). Instead, there is a task creation edge (n_{TC}, n_T).
- $E_S \subseteq N \times N \times K$ is the set of *task synchronization edges*. Given a task synchronization edge $(n_1, n_2, k) \in E_S$, n_1 is the *synchronized construct*, n_2 is the *synchronization point*, and k is the *kind of synchronization*, where $K = \{\text{strict}, \text{maybe}, \text{post}\}$ (The meaning of each kind is further explained in Section 3.2.1.1).

The PCFG is built in two steps. The first step is a conventional construction of a CFG with the singularities of the OpenMP support. Task nodes have no successor[3] at this stage because task synchronization edges are computed in the second step (explained in Section 3.2.1.1). $\mathcal{PCFG}_{simple}$ and $\mathcal{PCFG}_{struct}$ are the functions that build the graph. Both receive two parameters: the code to be represented (i.e., expression or statement), and the last node created in sequential order, n_{last} (n_{last} may be a list of nodes, e.g., the exit node of an *if*-statement has two last nodes, one from each branch of the statement). Furthermore, both methods return the node built, n_{ret}: $\mathcal{PCFG}_{simple}$ returns the only *simple* node node it creates, and $\mathcal{PCFG}_{struct}$ returns an *structured* node containing a graph. As an illustration, Figure 3.1 specifies some syntax-directed definitions of these functions representing those constructions that will be used more often in this document. For ease of reading, we do not specify the label for those *flow-edges* that are *unconditional*.

$$
\begin{aligned}
&\mathcal{PCFG}_{simple}(expr, n_{last}) = \langle\{n_{ret}\}, NULL, NULL, \{(n_{last}, n_{ret})\}, \varnothing, \varnothing\rangle\\
&\mathcal{PCFG}_{struct}(\textbf{if } (expr) \textbf{ then } stmt_1 \textbf{ else } stmt_2, n_{last}) =\\
&\qquad\qquad \langle\{n_1, n_2, n_3, n_{ret}\}, n_{EN}, n_{EX},\\
&\qquad\qquad \{(n_{EN}, n_1), (n_1, n_2, \text{t}), (n_1, n_3, \text{f}), (n_2, n_{EX}), (n_3, n_{EX}), (n_{last}, n_{ret})\},\\
&\qquad\qquad \varnothing, \varnothing\rangle\\
&\qquad\qquad \text{where } n_1 = \mathcal{PCFG}_{simple}(expr, n_{EN}),\\
&\qquad\qquad\qquad\quad n_2 = \mathcal{PCFG}_{struct}(stmt_1, n_1),\\
&\qquad\qquad\qquad\quad n_3 = \mathcal{PCFG}_{struct}(stmt_2, n_1)\\
&\mathcal{PCFG}_{struct}(\#\textbf{pragma omp task list_of_clauses } stmt_T, n_{last}) =\\
&\qquad\qquad \langle\{n_{TC}, n_T, n_{ret}\}, n_{EN}, n_{EX},\\
&\qquad\qquad \{(n_{EN}, n_{TC}), (n_{TC}, n_{EX}), (n_{last}, n_{ret})\},\\
&\qquad\qquad \{(n_{TC}, n_T)\}, \varnothing\rangle\\
&\qquad\qquad \text{where } n_T = \mathcal{PCFG}(stmt_T)\\
&\mathcal{PCFG}_{simple}(\#\textbf{pragma omp taskwait}, n_{last}) =\\
&\qquad\qquad \langle\{n_{ret}\}, NULL, NULL, \{(n_{last}, n_{ret})\}, \varnothing, \varnothing\rangle\\
&\mathcal{PCFG}_{simple}(\#\textbf{pragma omp barrier}, n_{last}) =\\
&\qquad\qquad \langle\{n_{ret}\}, NULL, NULL, \{(n_{last}, n_{ret})\}, \varnothing, \varnothing\rangle
\end{aligned}
$$

Figure 3.1: Examples of functions $\mathcal{PCFG}_{simple}$ and $\mathcal{PCFG}_{struct}$, which build the PCFG.

[3]The concepts of *predecessor* and *successor* have the usual definitions of graph theory, i.e., if a path leads from *x* to *y*, then *y* is said to be a successor of *x*, and *x* is said to be a predecessor of *y*.

3.2.1.1 Tasks synchronization data-flow algorithm

The second step of the PCFG construction is the computation of the synchronization edges. For that purpose, first we recognize the nodes that are able to synchronize tasks:

- *Task nodes* (n_T) synchronize previous sibling tasks whose dependences *match* (the next paragraph explains the algorithm that computes whether two tasks' dependences match).
- *Taskwait nodes* (n_{TW}) synchronize previous tasks that are child tasks of the current task.
- *Barrier nodes* (n_B) synchronize any previous task in the same binding region.
- *Virtual post-synchronization node* (n_{VPS}) is a unique node added to every PCFG that needs to virtually synchronize those tasks that may not be synchronized within the scope of the graph.

The methods involved in the tasks synchronization algorithm are described in Figure 3.2. There, given that an `inout` dependence is equivalent to an `out` dependence, a task T_2 *synchronizes* a task T_1 if the tasks are *siblings* and one of the following conditions fulfill: a) T_1 designates an `out` object that T_2 designates as `in` or `out` (RAW and WAW data hazards respectively), and/or b) T_1 designates an `in` object that T_2 designates as `out` (WAR data hazard). It may not be possible to statically determine if two tasks synchronize because it cannot be asserted if two dependences designate the same object (e.g., dependences of the form $var[expr]$). Thus, this process, modeled with function $\curlyvee$, can answer {*yes, no, unknown*}.

Consider N_T the set of nodes n_T in a given $PCFG$, and N_{deps} the maximum number of dependence clauses a task directive has. The cost of computing all synchronizations over that $PCFG$, which means calling $synchronizes$ for each pair of nodes in N_T, is $\mathcal{O}({N_T}^2 * {N_{deps}}^2)$.

$$match(d_1, d_2) = \begin{cases} YES, & \text{if } (d_1 = v_1 \land d_2 = v_2 \land v_1 = v_2) \\ & \qquad \lor (d_1 = v[k_1] \land d_2 = v[k_2] \land k_1 = k_2) \\ NO, & \text{if } d_1 = v_1[e_1] \land d_2 = v_2[e_2] \land v_1 \neq v_2 \\ & \qquad \land\ v_1, v_2 \text{ are arrays or restrict pointers} \\ UNK, & \text{otherwise} \end{cases}$$

$$a \curlyvee b = \begin{cases} YES, & \text{if } a = YES \lor b = YES \\ NO, & \text{if } a = NO \land b = NO \\ UNK, & \text{otherwise} \end{cases}$$

$$siblings(n_{T_1}, n_{T_2}) = \begin{cases} YES, & \text{if } n_{T_1} \land n_{T_2} \text{are } \textit{child tasks} \text{ of the same } \textit{task region} \\ NO, & \text{otherwise} \end{cases}$$

$$synchronizes(n_{T_1}, n_{T_2}) = siblings(n_{T_1}, n_{T_2}) \land \Big(\bigvee_{\substack{\forall d_1 \in \text{out}(n_{T_1}) \\ \forall d_2 \in \text{in}(n_{T_2}) \cup \text{out}(n_{T_2})}} match(d_1, d_2) \curlyvee \bigvee_{\substack{\forall d_1 \in \text{in}(n_{T_1}) \\ \forall d_2 \in \text{out}(n_{T_2})}} match(d_1, d_2) \Big)$$

Figure 3.2: Process that determines if two tasks synchronize.

Synchronization edges have a kind k that may take one of the following values:

- *strict*: a *task node* n_{T_1} certainly synchronizes in a node n because either:
  - $n = n_{TW}$ and both are in the same binding region.
  - $n = n_B$ and n is a region that encloses, or is the same region as, the binding region of n_{T_1}.
  - $n = n_{T_2}$ and $synchronizes(n_{T_1}, n_{T_2}) = YES$.
- *maybe*: a *task node* n_{T_1} cannot be statically decided to synchronize with n_{T_2} (i.e., $synchronizes(n_{T_1}, n_{T_2}) = UNK$)
- *post*: the synchronization may occur any time after the function ends.

Synchronization edges are computed using a forward data-flow algorithm that defines the tasks live at the entry point, $LITask \subseteq N$, and the exit point, $LOTask \subseteq N$, of each node in a PCFG.

A task node $n_T \in LITask(n)$ if:

$$n_T \in ancestor(n) \wedge$$
$$\nexists\, n' \in predecessor(n) : e = (n_T, n', strict) \in E_S(n')$$

A task node $n_T \in LOTask(n)$ if:

$$synchronizes(n_T, n) = \{NO, UNK\} \vee$$
$$\text{all matched dependences in } n \text{ are inputs} \vee$$
$$n_T \text{ has unmatched dependences}$$

Additionally, when computing the $LOTask$ set, those tasks that remain alive because all target's matched dependences are inputs are singled out. These tasks may be the source dependence of several target tasks with input dependences on the same variables, and definitely synchronize when a taskwait or barrier is reached.

Theorem 1 $TS_{DFAF} = \langle \mathcal{L}, \mathcal{T}_f \rangle$ *is the bounded monotone forward* Tasks Synchronization Data-Flow Algorithm *that computes the task synchronizations over a graph G, and consists of:*

- $\mathcal{L} = \langle S \times R, \sqcap \rangle$ *is the meet-semilattice[8] that imposes a partial order over all possible data-flow values in the algorithm, where:*
  - $S \subseteq \{n_T \in N\}$ *is a subset of all task nodes, with two special elements:* $\top$*, the lattice top element equivalent to the empty set, and* $\bot$*, the lattice bottom element equivalent to* S*.*
  - $R = N \times KIND'$*, where* $KIND' = \{strict, maybe\}$*, is the set of kind relationships of two synchronized nodes.*
  - $\sqcap = (\cup, \varnothing)$ *is the meet operator that merges flow values and imposes an order over the lattice by using just the first element in the pair representing each data-flow value.*
    The meet operator is used to compute the live tasks at the entry of a node $n \in N$ *as follows:*

$$LITask(n) = \Big(\bigcup_{p \in pred(n)} LOTask(p), \varnothing\Big)$$

The meet operator is monotone. Given the elements x_1*,* x_2*,* y_1 *and* y_2*, it fulfills:*

$$x_1 \sqsubseteq y_1 \wedge x_2 \sqsubseteq y_2 \Rightarrow (x_1 \sqcap x_2) \sqsubseteq (y_1 \sqcap y_2)$$

– *$\mathcal{T}_f = \{f : S \times R \to S \times R\}$ is the family of transfer functions that maps the program behavior onto $\sqcap$ computing $LOTask(n)$ for each $n \in N$ as follows:*

$$
\begin{aligned}
f(n_T) = (&\{n_T{}' | n_T{}' \in LITask(n_T) \wedge (\neg siblings(n_T, n_T{}') \\
&\quad \vee synchronizes(n_T, n_T{}') \neq YES)\}, \\
&\{(n_T{}', \boldsymbol{strict}) | n_T{}' \in LITask(n_T) \\
&\quad \wedge siblings(n_T, n_T{}') \wedge synchronizes(n_T, n_T{}') = YES\} \\
&\cup \{(n_T{}'', \boldsymbol{maybe}) | n_T{}'' \in LITask(n_T) \\
&\quad \wedge siblings(n_T, n_T{}'') \\
&\quad \wedge synchronizes(n_T, n_T{}'') = UNK\}) && \text{/*task*/} \\
f(n_{TW}) = (&\{n_T | n_T \in LITask(n_{TW}) \wedge \neg siblings(n_{TW}, n_T)\}, \\
&\{(n_T, \boldsymbol{strict}) | n_T \in LITask(n_{TW}) \\
&\quad \wedge siblings(n_{TW}, n_T)\}) && \text{/*taskwait*/} \\
f(n_B) = (&\varnothing,\ \{(n_T, \boldsymbol{strict}) | n_T \in LITask(n_B)\}) && \text{/*barrier*/} \\
f(n) = (&\{n_T | n_T \in LITask(n)\}, \varnothing) && \text{/*any other node*/}
\end{aligned}
$$

All transfer functions are monotonic. Given the elements x and y, they fulfill:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Each transfer function computes the pair $\langle LOTask(m), SynchronizedTask(m)\rangle$ of a given node m. The first element is the set of tasks that are still live after the execution of m. The second element is the set $S \times R$ of tasks synchronized in m. E.g., for a task node n_T, the transfer function $f(n_T)$ returns a pair where: a) the first element contains those tasks $n_{T'}$ in the set $LITask(n_T)$ that, either are not siblings of n_T, or are not synchronized in n_T ($synchronizes(n_T, n_{T'}) \neq YES$), or b) the second element contains those tasks in the set of $LITask(n_T)$ that are siblings of n_T and are certainly synchronized in n_T ($synchronizes(n_T, n_{T'}) = YES$).

The semi-lattice $\mathcal{L}$ is monotone and of finite height (the number of tasks in a program is finite, thus, the number of sets with the different combinations of these tasks is finite). Because of that, the algorithm is guaranteed to converge.

Algorithm 1 shows the high-level iterative algorithm that computes the tasks synchronizations over a PCFG G. The algorithm initializes the root node of the graph with the lattice least upper bound, $\top$. Then, it performs forward traversals over G, computing the $LITask(n)$ and $LOTask(n)$ sets of each node n, until no data-flow value changes. At this point there may still be live tasks at the exit node of G, which shall be synchronized with the virtual post-synchronization node, n_{VPS}, of the graph.

Algorithm 1 High-level algorithm for synchronizing tasks within a PCFG.

1: $LITask(n_{EN}) = LOTask_{new}(n_{EN}) = \top$
2: **for each** $n \in N$ $LOTask_{new}(n) = \top$ **do**
3: $worklist = p — p \in succ(n_{EN})$
4: **while** !$worklist$.empty() **do**
5: $worklist = worklist$ - n
6: $LITask(n) = \bigcup_{p \in pred(n)} LOTask_{new}(p)$
7: $LOTask_{old}(n) = LOTask_{new}(n)$
8: $LOTask_{new}(n) = f(n)$
9: **if** $LOTask_{old}(n) \neq LOTask_{new}(n)$ **then**
10: $worklist = worklist \cup s — s \in succ(n)$
11: **end if**
12: **end while**
13: **for each** $n_T \in LOTask(n_{EX})$ **do**
14: add_edge(n_T, n_{VPS}, $post$, NULL) to G
15: **end for**
16: **end for**

As an illustration, Figure 3.3 shows a simplified version of the PCFG resulting from the code in Listing 3.1, a blocked matrix multiplication using OpenMP tasks. The information related to the tasks is drawn in red (*task* and *task creation* nodes, and *synchronization* edges with their corresponding labels). Note the synchronization edge from the task to the task itself tagged as *Maybe* because the inout dependence on $C[i : BS][j : BS]$ cannot be statically decided at this point, as its value may vary between task instances (A and B are not considered to compute this edge because both are input dependences. Furthermore, the task escapes its scope because there is no synchronization, so it is connected to the virtual post-synchronization node.

```
1 void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N]) {
2    for (int i = 0; i < N; i+=BS)
3       for (int j = 0; j < N; j+=BS)
4          for (int k = 0; k < N; k+=BS)
5             #pragma omp task private(ii, jj, kk) \
6                              depend(in: A[i:BS][k:BS], B[k:BS][j:BS]) \
7                              depend(inout: C[i:BS][j:BS])
8                for (int ii = i; ii < i+BS; ii++)
9                   for (int jj = j; jj < j+BS; jj++)
10                     for (int kk = k; kk < k+BS; kk++)
11                        C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
12 }
```

Listing 3.1: Matrix multiplication using OpenMP tasks (Example *task_dep.5.c* from the specification examples [112])

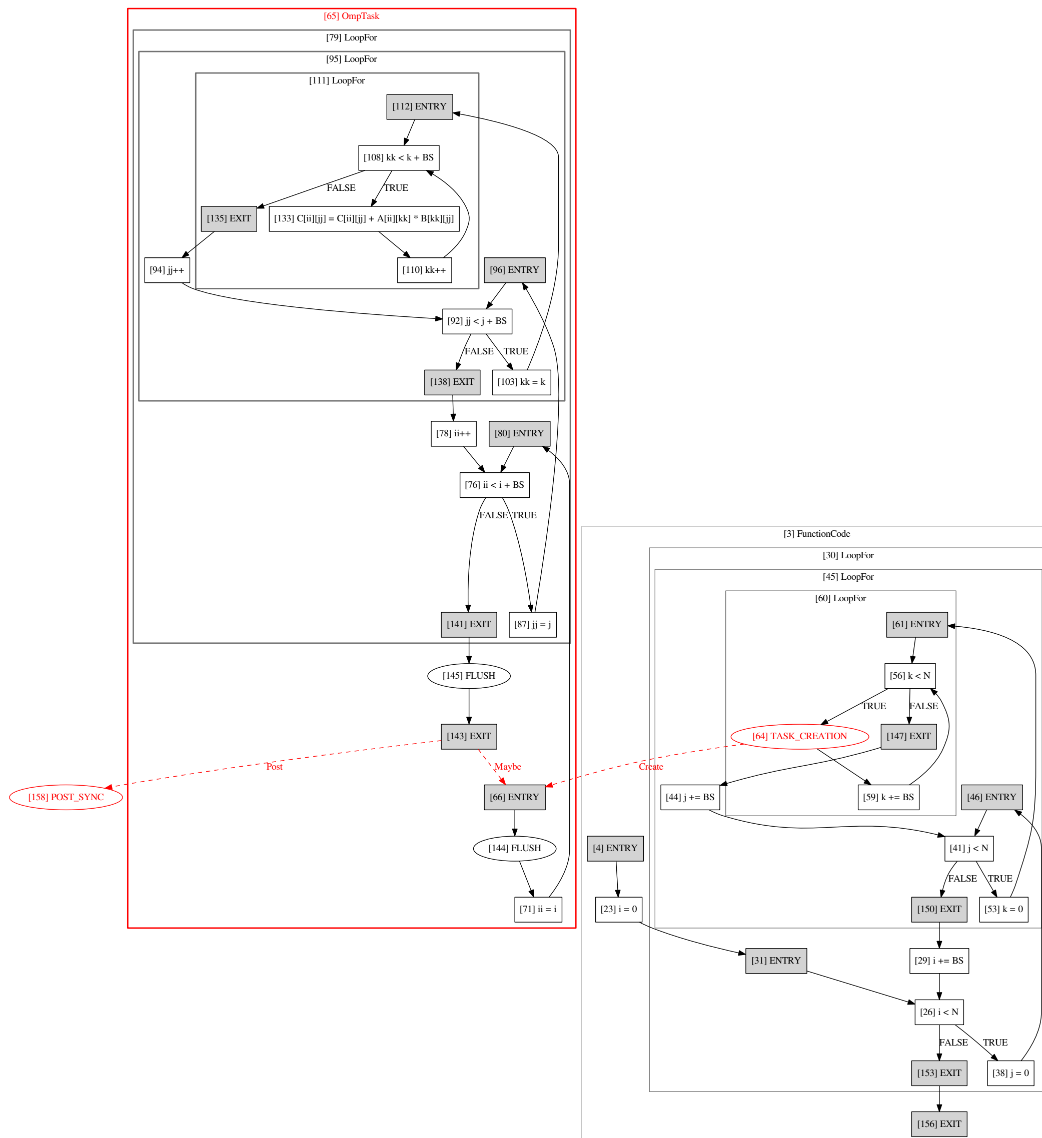


Figure 3.3: PCFG for code in Listing 3.1.

3.2.2 Use-Definition analysis

Use-definition is an inter-procedural context-sensitive[4] analysis that computes the variables that are used and defined at each point of a program. This means that, for each node n in the PCFG, we compute the following sets:

- *UE(n)* is the set of upwards exposed variables[5].
- *Kill(n)* is the set of variables which have at least one write access within the node.
- *Undef(n)* is the set of variables which use cannot be determined, e.g., the pointed value of a pointer passed as argument to a function which code is not reachable at compile-time.

[4]*Context-sensitive analysis* is an inter-procedural analysis that considers the calling context when analyzing the target of a function call.

[5]An *upwards exposed* variable is that whose first use is a read.

The rules that classify the variables are the following: a) if a variable is first classified as *UE*, it can also be classified as *Kill* (if it is first read and then written), or be reclassified as *Undef* (if at some point the access cannot be determined), b) if a variable is first classified as *Kill*, it cannot be later classified as *UE* or *Undef*, c) if a variable is first classified as *UE* and *Kill*, it cannot later be classified as *Undef*, and d) if a variable is first classified as *Undef*, it cannot be later reclassified.

The forward data-flow algorithm that computes use-definition information works from top to bottom, regarding the control flow, and from inside to outside, regarding the topology of the graph (the PCFG is a graph where some nodes -*structured* nodes- are graphs themselves). Generally, usage information is propagated to *structured* nodes following a backwards traversal of the inner nodes. For each inner node, the information of its children is merged with the information computed for its own statements using the following equations:

$$UE_{merge}(n) = \bigcup_{m \in Successor(n)} UE(m) - Kill(n)$$

$$Kill_{merge}(n) = \bigcup_{m \in Successor(n)} Kill(m) - Undef(n)$$

$$Undef_{merge}(n) = \bigcup_{m \in Successor(n)} Undef(m) - \{v | v \in UE(n) \wedge v \in Kill(n)\}$$

In the last step of the propagation process, when the information computed for the last inner node (the entry node of the structured node) is propagated to the structured node, the visibility of the variables is taken into account. In that sense, the algorithm considers the context where the variables are declared, and the data-sharing attributes in case of evaluating a node representing an OpenMP node. Hence, variables local to the structured node, and private variables (including private and firstprivate for the *Kill* and the *Undef* sets, and only private and lastprivate for the *UE* set) are removed from the final sets, and all firstprivate variables are included in the *UE* set. The use of private variables refers indeed to different symbols from the use of the original variables.

Additionally, the asynchronism introduced by the OpenMP tasking model requires considering the propagation of the information computed for task nodes in a different way. Consider the code shown in Figure 3.2. Since the three tasks can execute in any order, the usage computation for function foo must note that although x is undefined in $Task3$, it is both upwards exposed and killed in $Task1$, so it can be removed from the *Undef* set. Conversely, y is undefined in $Task3$, and only killed in $Task1$. The compiler cannot decide whether the variable is also upwards exposed, and thus it can be removed from the *Kill* set to remain only in the *Undef* set.

Use-definition is inter-procedural, so the analysis of a called function is propagated to wherever it is called when possible (although Mercurium does not have the utilities for whole program analysis, hence we can only perform Interprocedural Analysis (IPA) for methods contained in the same source file, this is not a problem because an application can always be

```
1 void bar(int&);
2
3 int x=0;
4 int y;
5
6 void foo() {                    // UE: x, Kill: x, Undef: y
7     #pragma omp task            // Task1. UE: x, Kill:x,y
8     {
9         #pragma omp task        // Task2. Kill: x
10            x = 1;
11        y = x;
12    }
13    #pragma omp task            // Task3. Undef: x,y
14        bar(x);
15    #pragma omp taskwait
16 }
```

Listing 3.2: OpenMP example illustrating the propagation of usage information to outer nodes.

analyzed by embedding all the code in the same file). The compiler also defines the behavior of common C standard library methods, so when they are called, their behavior is propagated.

Use-definition analysis is a previous step for many other data-flow analyses such as liveness and reaching definitions. The following sections introduce these analyses.

3.2.3 Liveness

Liveness analysis is a data flow analysis that computes, for each program point, the variables that may be potentially read before their next write. Therefore, a variable is live if it holds a value that may be needed in the future. This means that, for each node n in the PCFG, we compute the following sets:

- *LI(n)* is the set of the variables that are live at the entry of node n.
- *LO(n)* is the set of variables that are live at the exit of node n.

The backward data-flow algorithm that performs liveness analysis works from bottom to top, regarding the control flow, and from inside to outside, regarding the topology of the graph (relevant for *structured* nodes). Figure 3.4 shows the equations that compute the *LI* and *LO* sets of a given node n. The computation depends on the type of node (*simple* or *structured*) and on the OpenMP semantics (if applicable). Hence, for *simple* nodes, we use the common data-flow equations for defining liveness, which use the upper exposed variables, *UE(n)*, and the defined variables, *Kill(n)*. And for *structured* nodes, we propagate the information computed in the inner nodes (concretely, the entry node n_{EN} of n) to the outer node. Furthermore, for structured nodes representing OpenMP constructs, we take into account the visibility of the variables. Accordingly, private and lastprivate variables are not propagated to outer nodes when computing the *LI* set, and private and firstprivate are not propagated to the outer node when computing the *LO* set.

In addition to the visibility of the variables, liveness analysis must also take into account the asynchronism introduced by the OpenMP tasking model, and how this affects to the PCFG representation. In that sense, when a task is encountered within a loop, we add the task itself to the list of successors of the task (in case it was not there due to dependence expressions), because a task instance could use data produced in other task instances.

$$LI(n) = \begin{cases} UE(n) + (LO(n) - Kill(n)), & \text{if } n \text{ is } simple \\ LI(\mathcal{PCFG}(n) \rightarrow n_{EN}) - (Private(n) \cup Lastprivate(n)), & \text{if } n \text{ is } structured \end{cases}$$

$$LO(n) = \begin{cases} \bigcup\limits_{m \in Successor(n)} LI(m), & \text{if } n \text{ is } simple \\ LO(\mathcal{PCFG}(n) \rightarrow n_{EX}) - (Private(n) \cup Firstprivate(n)), & \text{if } n \text{ is } structured \end{cases}$$

Figure 3.4: Equations that determine the liveness attributes of a PCFG node.

This analysis is inter-procedural at the same level as use-definition chains. This means the same limitations apply to the results of liveness analysis, thus IPA is only possible for methods contained in the same source file.

3.2.4 Reaching definitions

Reaching definitions is a data-flow analysis that determines which definitions may reach a given point in the code. This means that, for each node n in the PCFG, we compute the following sets:

- *RI(n)* is the set of definitions reaching the entry point of node n.
- *RO(n)* is the set of definitions reaching the exit point of node n.

We compute reaching definitions over the PCFG following a common iterative forward data-flow algorithm that traverses the graph from top to bottom, regarding the control flow, and from inside to outside, regarding the topology of the graph (relevant for *structured* nodes). Figure 3.5 shows the equations that compute the ***RI*** and ***RO*** sets of a given node n. These equations depend on the type of node (simple or structured). Hence, for simple nodes, we use the common data-flow equations for defining reaching definitions, which use the set of generated (declared and initialized) variables, *Gen(n)*, and the set of defined variables, *Kill(n)*. And for structured nodes, we propagate the information computed in the inner nodes (precisely, the entry node n_{EN} of n for the RI set, and the exit node n_{EX} of n for the RO set) to the outer node.

$$RI(n) = \begin{cases} \bigcup\limits_{m \in Predecessor(n)} RO(m), & \text{if } n \text{ is } simple \\ RI(\mathcal{PCFG}(n) \rightarrow n_{EN}), & \text{if } n \text{ is } structured \end{cases}$$

$$RO(n) = \begin{cases} Gen(n) \bigcup (RI(n) - Kill(n)), & \text{if } n \text{ is } simple \\ RO(\mathcal{PCFG}(n) \rightarrow n_{EX}), & \text{if } n \text{ is } structured \end{cases}$$

Figure 3.5: Equations that determine the reaching definitions of a PCFG node.

The asynchronism introduced by the OpenMP tasking model requires special attention because statements within a task can give rise to definitions that reach points out of the regular control flow. For this reason, it is necessary to compute all regions of code that are concurrent with a task [129],

and propagate the reaching definitions across those regions of code. The code in Listing 3.6a and the simplified version of its corresponding PCFG in Figure 3.6b show an example of this situation, where the regular control flow traversal is not enough to correctly compute reaching definitions: definition of res in $Task1$ may reach $Task2$, and definition of res in $Task2$ may reach $Task1$.

Reaching definitions are used to analyze loops, particularly induction variables and their boundaries. This information is later used for optimizations out of the scope of this thesis, such as user-directed vectorization [37], and transformations such as the static generation of task dependency graphs (detailed in Chapter 5).

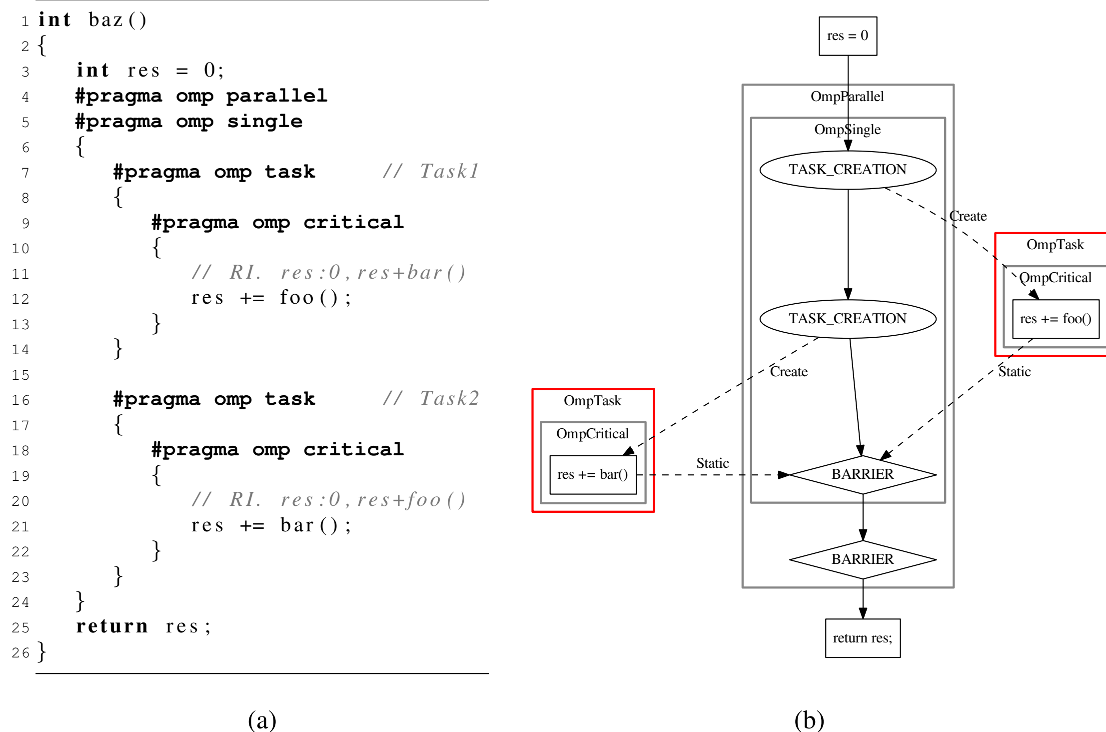


Figure 3.6: Example illustrating the impact of OpenMP tasks regarding reaching definitions: (a) code snippet (b) simplified PCFG.

3.3 Impact

The works developed in the context of classic compiler analysis for OpenMP have been used in the doctoral thesis of Caballero [36] to develop a user-directed vectorization infrastructure aimed at improving the exploitation of SIMD instructions with OpenMP. The results of this work, in turn, have been used to define the SIMD extensions included in version 4.0 of the OpenMP specification.

Additionally, under the umbrella of the TERAFLUX European project [55, 148], Patejko analyzed the properties of masks used in vector instructions (so-called Mersenne masks) using our analysis framework, particularly, the PCFG and liveness analysis.

3.4 Conclusion

Classic compiler techniques are incredibly powerful to both compiler optimization and correctness checking. However, these techniques require some revision if they are to be used with languages expressing parallelism. This chapter addresses the adaptation of four fundamental compiler analysis algorithms to incorporate parallel semantics, including control-flow analysis by means of a PCFG adapted to the OpenMP tasking model, and data-flow analysis by means of use-definition, liveness and reaching definition analyses built on top of the PCFG. The four algorithms presented here create the basis for the rest of the work presented in this thesis.

4

Correctness in OpenMP

This section presents a series of analyses aimed at tackling correctness in the OpenMP tasking model. As a foreword, we briefly present the contributions presented in the context of the Master's thesis that precedes this PhD thesis. Then, we identify a set of cases that users should be aware of because they may lead different problems (loss of performance or race conditions). After that, we present a set of analyses based on the framework presented in Chapter 3 and implemented in the Mercurium compiler that can supply hints about errors that may occur at run-time for the presented cases. Finally, we test the mechanism with several students and benchmarks, and also compare our results with those of Oracle Solaris Studio 12.3 compiler [114].

4.1 Contributions of the M.S. thesis

During the preceding Master's thesis [130] we started our research about high-level compiler analysis and its application to correctness. We developed a primary version of the classic analyses (i.e., CFG, use-definition and liveness analyses) in an old version (1.3) of the Mercurium compiler (see Section 2.2.1 for further details on the current implementation of Mercurium). On top of that, we implemented two algorithms to automatically determine some clauses of the OpenMP and OmpSs task constructs: the data-sharing clauses, and the dependence clauses. The basics of these algorithms are introduced in this section.

4.1.1 Automatic scope of variables

All variables appearing within an OpenMP construct have default data-sharing as defined in the specification (either predetermined or implicitly determined, see Section 2.15.1 of the specification [113] for more details). Nonetheless, users usually need to explicitly scope most of these variables changing the default data-sharing values in order to ensure the correctness of their codes (e.g., avoiding data race conditions) and enhance their performance (e.g., privatizing shared variables).

We proposed an algorithm to automatically determine the data-sharing attributes of any OpenMP task triggered by a new keyword `AUTO` attached to the task's clause `default` [129]. The algorithm determines the regions of code that are concurrent with a given task and defines the

data-sharing attributes based on two factors: a) the usage of the variables in all concurrent regions, and b) their liveness properties after the execution of the task.

The algorithm is perfectly accurate: it neither reports negatives nor false positives. However, the algorithm is limited to the visibility of the concurrent code at compile-time. Thus, specific rules cover the cases where the algorithm cannot determine the data-sharing attribute of a variable, and the undetermined variables are reported back to the user for manual scoping.

4.1.2 Automatic detection of task dependences

OpenMP implements a fine-grain synchronization mechanism that enables the data-flow driven execution of tasks by means of data-dependence clauses. These dependences regulate the generation of a TDG that represents the order in the tasks that cannot be broken, and honors the semantics of the values allowed for these clauses: `in`, `out` and `inout`.

We proposed an algorithm to automatically determine the data-dependence attributes of any OpenMP task triggered by the new keyword `AUTO_DEPS` attached to the task's clause `default` [131]. The algorithm works in 3 steps: 1) define the regions of code that run concurrently with a given task, 2) compute the data-sharing attributes of all involved variables, and 3) compute the data-dependence attributes for all variables determined as shared in step 2, based on possible races and liveness properties.

The algorithm is perfectly accurate, however it is limited to the visibility available at compile-time. Thus, specific rules cover the cases when the algorithm cannot determine the data-sharing attributes or the portions of code that are concurrent with a given task. In these cases, undetermined variables are reported to the user to manually define the dependence clauses.

4.2 Related work

Despite the flexibility and programmability delivered by OpenMP, the language introduces some difficulties of its own. Süß and Leopold described fifteen OpenMP mistakes typical of novel programmers [146] in the context of OpenMP 3.0. They classified these mistakes in two groups: those regarding correctness (e.g., unprotected access to shared variables and variables improperly privatized), and those regarding performance (e.g., use of a critical construct that could be replaced by an atomic construct to improve efficiency, and unnecessary flushes). Later, Münchhalfen et al. [102] further classified OpenMP errors considering the tasking and the accelerator models. They divide the possible errors in two groups: a) *defects*, which are programming errors (i.e., incorrect source code) and include non-conforming programs (e.g., uninitialized locks, and invalid nesting of regions) and conceptual defects (e.g., locks as barriers, and missing data mapping to the accelerator), and b) *failures*, which are error manifestations (e.g., execution abortion and deadlocks) and include situations such as race conditions and deadlocks.

Several approaches have been proposed for OpenMP run-time correctness checking. Li et al. [85] presented a tool based on a hybrid methodology involving on-line testing (comparing the results of serial and parallel executions) and off-line testing (recording the values of the

variables at the entry and the exit of parallel constructs of a serial execution, and comparing these values with those of parallel executions). Ha et al. [61] used a hybrid technique that combines *happens-before* analysis and *lockset* analysis for efficiently detecting data races at run-time. There are also production tools available for OpenMP and other parallel languages, such as Intel® Thread Checker [117], TotalView [39] debugger, and the Valgrind based tool Helgrind+ [74].

The common aspect of all the tools and methodologies described above is that they work at run-time, as they require the execution of the program to detect errors. The main benefit of using a runtime tool is that all variables have a known value, thus no disambiguation processes are necessary (e.g., alias analysis). Unfortunately, this approach requires users to execute their programs along with the runtime tools in order to find errors and, as a result, some overhead is introduced in the execution. Additionally, there is no guarantee that the error will occur in that particular execution, as well as compiler optimizations may hide some errors (e.g., variables stored in registers may hide race conditions). A few compile-time techniques aiming at checking OpenMP correctness have been published as an alternative to runtime tools. Lin [87] described a CFG and a region tree used to statically detect non-concurrent blocks of code and race conditions in OpenMP 2.5 programs with the Sun Studio 9 Fortran compiler. Basupalli et al. [27] presented *ompVerify*, a tool based on the polyhedral model that is able to detect several errors in OpenMP parallel loops.

4.3 Automatic solution of common mistakes involving OpenMP tasks

Different mistakes in the use of OpenMP tasks may lead to run-time errors, non-deterministic results or loss of performance. The reasons that cause these problems are mainly:

1. Bad synchronization of the tasks, either using synchronization directives (e.g., barrier and taskwait), or task dependence clauses.
2. Bad usage of the variables involved in the construct due to a bad specification of the data-sharing attributes.

The following sections describe the set of case studies we have identified. For each case, a simple code snippet illustrates the scenario, and a simplified version of its PCFG accompanies the algorithm that allows discovering correctness issues. All these methods are also adapted to the OmpSs programming model.

4.3.1 Variables' storage

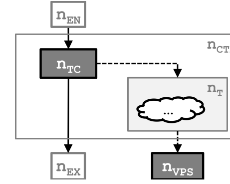
Scenario Variables with automatic storage duration are those that are allocated and deallocated automatically when the program flow enters and leaves the enclosing code block. When such a variable is shared in a task, the task shall be executed before the block of the variable ends. Otherwise the variable may not be accessible any more and a run-time error may occur.

Example This scenario is illustrated in Figure 4.1a, where variable *a* is local to function *foo* and its storage will disappear once this function exits. Its data-sharing is explicitly determined to be shared, so the task will access the original block of storage. If the task is deferred until the function

returns, *a* will have disappeared. In the corresponding PCFG, Figure 4.1b, the task is synchronized in n_{VPS}, a virtual post-synchronization node indicating the task may scape the function.

```
1 void foo()
2 {
3   int a[1000000] = {0};
4   #pragma omp task shared(a)
5   { ... }
6 }
```

(a) code snippet



(b) simplified PCFG

Figure 4.1: Scenario illustrating an automatic storage variable that may be accessed after its lifetime ends.

Compiler analysis Algorithm 2 shows the code that, given a complete PCFG, determines the variables that may be accessed after their storage lifetime has ended. For each task node, the method gathers the nodes where the task synchronizes, $synclist$. Then, for each memory location (consider also array subscripts, class member accesses and dereferences) accessed as shared within the task, v_S, the method looks for the context node n_{CTX} where the variable is declared, and checks whether n_{CTX} contains all nodes in $synclist$. Variable v_S will potentially be wrongly accessed if at least one node in $synclist$ is not contained in n_{CTX}.

Algorithm 2 High-level algorithm to detect accesses to variables in OpenMP tasks whose lifetime may have ended.

1: $result = \varnothing$
2: **for each** $n_T \in N$ **do**
3: $synclist = \{n \in N$: e=$(n_T, n, KIND)\}$
4: **for each** $v_S \in shared(n_T)$ **do**
5: n_{CTX} = context node where v_S is declared
6: **if** (n_{CTX} is global context) **then**
7: **break**
8: **end if**
9: **for each** $n \in synclist$ **do**
10: **if** n_{CTX} does not contain n **then**
11: $result = result \cup v_S$
12: **break**
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: **return** $result$

Applied to the example in Figure 4.1a and according to Figure 4.1b, the context node n_{CTX} of *a* does not contain the virtual post-synchronization node n_{VPS} where the task n_T synchronizes, so *a* may be accessed after it is deallocated.

Compiler solution To avoid this situation the compiler proposes one of the following solutions:

- Change the data-sharing attribute of the variable to `private` or `firstprivate`: suitable for basic data types (integer, floating point and pointer).
- Introduce a `taskwait` before the enclosing block ends: suitable for arrays and structures, since privatizing them may lead to a loss of performance.

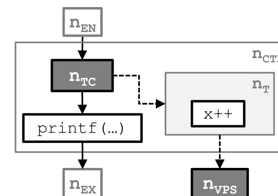
4.3.2 Data-race conditions

Scenario A data race occurs when two or more threads access shared data and at least one of the accesses is a write. Tasks accessing shared variables must synchronize the accesses in such a situation, otherwise there exists a race condition.

Example We illustrate this case in Figure 4.2a, where variable *x* is implicitly determined as shared. No synchronization assures the post-increment of *x* is executed before the call to the function *printf*, and thus the result of this code is non-deterministic.

```
1 int x = 0;
2 void foo()
3 {
4   #pragma omp task    // T
5     x++;
6   printf("x=%d\n", x);
7 }
```

(a) code snippet



(b) simplified PCFG

Figure 4.2: Scenario illustrating a race condition due to a wrong synchronization of a task.

Compiler analysis Algorithm 3 shows the code that, given a complete PCFG, determines the variables in a race situation. For each task node, the algorithm first computes all regions in the task enclosing function that are concurrent with the task, considering other tasks and sequential code. This first step is an extension of a previously developed method to automatically determine data-sharing attributes [129], and takes into account different aspects:

- The storage of variables: global variables, dynamic storage locations and reference parameters may be used outside the function.
- The data-sharing attributes of the OpenMP constructs enclosing the task: variables which are private in a region cannot cause a race with uses outside that region.
- The outermost iterative statement enclosing the task: different instances of the task may run concurrently, and the task instances may be concurrent with different iterations of the code inside the loops that enclose the task construction.
- The points where the task may be synchronized: code after these points cannot be concurrent.

Once all concurrent code has been identified, for each v_S, shared memory access, the algorithm gathers the nodes in the concurrent regions using v_S and the nodes in the task using v_S, and checks that at least one of those accesses is a write. If so, the method checks whether all accesses are synchronous (protected in a `critical` or `atomic` construct) and, if not, the variable is reported as a race. Variables that cannot be determined to be in a race condition because their storage outlives the function are also reported.

Algorithm 3 High-level algorithm to detect race conditions in OpenMP tasks.

1: $true_race_list = maybe_race_list = \varnothing$
2: **for each** $n_T \in N$ **do**
3: $\quad R = \{n \in N : n \in concurrent_regions(n_T)\}$
4: $\quad$ **for each** $v_S \in shared(n_T)$ **do**
5: $\quad\quad U_T = \{u_T \in N : u_T$ is inner node of $n_T \wedge u_T$ uses $v_S\}$
6: $\quad\quad U_R = \varnothing$
7: $\quad\quad$ **for each** $r \in R$ **do**
8: $\quad\quad\quad U_R = \{u_R \in N : u_R$ is inner node of $R \wedge u_R$ uses $v_S \wedge u_R \notin U_T\} \cup U_R$
9: $\quad\quad\quad$ **if** $\exists u_R \in U_R, \exists u_T \in U_T$: $(u_R$ is write $\vee$ u_T is write$)$
$\quad\quad\quad\quad \wedge$ $(u_R$!is synchronous $\vee$ u_T !is synchronous$)$ **then**
10: $\quad\quad\quad\quad true_race_list = true_race_list \cup v_S$
11: $\quad\quad\quad\quad$ **break**
12: $\quad\quad\quad$ **end if**
13: $\quad\quad\quad$ **if** v_S is global $\vee$ v_S is dynamic storage $\vee$ v_S is reference parameter **then**
14: $\quad\quad\quad\quad maybe_race_list = maybe_race_list \cup v_S$
15: $\quad\quad\quad$ **end if**
16: $\quad\quad$ **end for**
17: $\quad$ **end for**
18: **end for**
19: **return** $true_race_list, maybe_race_list$

Applied to code in Figure 4.2a and according to Figure 4.2b, there is no task concurrent with the task T. Only sequential code between the task creation and the exit node is concurrent with T (we use the exit node because the task synchronizes in a virtual post-synchronization node, meaning that the synchronization occurs sometime after the function). So, there is a concurrent usage of *x*, one read and one write, and no access is synchronous, hence a race condition exists. Note that, if there was no call *printf*, there will still be a possible race on *x*, because it is a global variable and might be used outside the function concurrently with the task.

Compiler solution To avoid this situation, the compiler proposes two solutions:

- Insert a `taskwait` between the two uses.
- Protect the accesses to the variables with `critical` or `atomic` constructs.

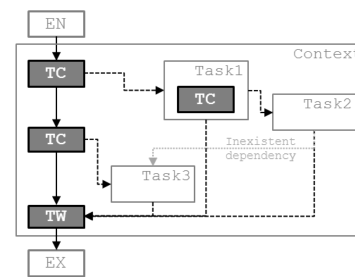
4.3.3 Dependences among non-sibling tasks

Scenario OpenMP allows the definition of task dependences only among sibling tasks (i.e., tasks that are children of the same task region). Other synchronization constructs such as barrier or taskwait must be used to impose an order between tasks created in different regions.

Example Consider the code in Figure 4.3a and its corresponding PCFG in Figure 4.3b. This example defines *Task1* and *Task3* in the region of the *initial task*[1], and *Task2* in the explicit task region associated with *Task1*. We define $TR(n)$ as the inner most enclosing node n_T that contains node n. In the example, $TR(n_{Task1}) = TR(n_{Task3}) = \mathscr{G}$, and $TR(n_{Task2}) = n_{Task1}$. Therefore, dependences of *Task2* are not checked with dependences of *Task3*, because the tasks belong to different task regions. As a consequence, the result is non-deterministic and depends on the order of execution of *Task2* and *Task3*.

```
1  void foo(int x) {
2    #pragma omp task                          // Task1
3    {
4      #pragma omp task depend(out:x)          // Task2
5        { ... }
6    }
7    #pragma omp task depend(in:x)             // Task3
8        { ... }
9    #pragma omp taskwait
10 }
```

(a) code snippet



(b) simplified PCFG

Figure 4.3: Scenario illustrating a useless definition of dependences between non-sibling tasks.

Compiler analysis Algorithm 4 shows the code that, given a complete PCFG, discovers tasks that may define dependences among non-sibling tasks. It takes into account only nested tasks n_T with dependence clauses and at least one outgoing edge whose target is not contained in $TR(n_T)$. For each of such tasks, the algorithm gathers the dependence clauses of n_T and the dependence clauses of all those tasks defined after n_T in any context enclosing $TR(n_T)$. Then, we use the previously defined $synchronizes$ method to check whether the two tasks could synchronize. If the result is not NO,then n_T is reported to have dependences with a non-sibling task.

Applied to the example in Figure 4.3a and according to Figure 4.3b, nested *Task2* has matching dependences with *Task3*, which is defined after *Task2* in a context containing $TR(n_{Task2})$.

Compiler solution To avoid such an scenario, the compiler proposes two possible solutions:

- Synchronize the nested task with a `taskwait` before its task region ends (i.e., within its parent task).
- Propagate the dependences from the nested task to its parent task.

[1]The *initial task* is an implicit task associated with the inactive implicit parallel region surrounding the whole OpenMP program. It completes at program exit.

Algorithm 4 High-level algorithm to detect dependences among non-sibling OpenMP tasks.

```
 1: result := ∅
 2: for each n_T ∈ N do
 3:   if n_T has dependence clauses ∧ ∃e = (n_T, n, KIND) : e ∉ TR(n_T).E_S then
 4:     T := tasks defined after n_T within any context enclosing TR(n_T)
 5:     for each n_T' ∈ T do
 6:       if synchronizes(T, T') then
 7:         result := result ∪ (n_T, n_T')
 8:         break
 9:       end if
10:     end for
11:   end if
12: end for
13: return result
```

4.3.4 Incoherent data-sharing

Scenario As explained in Section 2.1.1.2, OpenMP defines rules to determine the data-sharing attributes of any construct. Users have to understand these rules to resolve each default attribute, as well as to explicitly the attributes for those variables whose default value is not correct. Both mechanisms are error-prone due to the large amount of variables that may be involved in each task. In that sense, the compiler can check the following incoherences regarding data-sharing attributes:

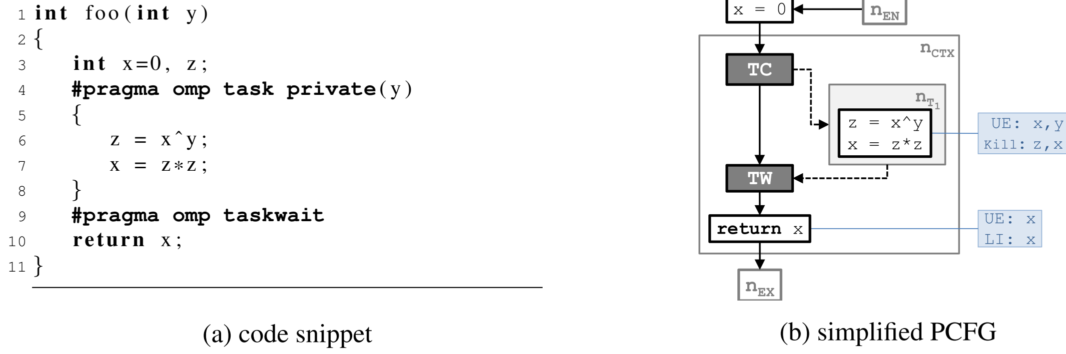
Incoherent Dead: variables defined within a task and never used in that task, but used after the task synchronization, should be shared.

Incoherent Private: upwards exposed variables in a task should not be private.

Incoherent Firstprivate: variables that are not upwards exposed within a task should be either private or shared (depending on the liveness of the variable and the chances of a data race).

Example Figure 4.4a illustrates an example of all these situations: variable *x* is implicitly determined as firstprivate and thus the modifications to this variable will not be visible after the task; variable *y* is explicitly determined as private, therefore its value inside the task is undefined at compile-time; and variable *z* is implicitly determined as firstprivate but its initial value is never read. Figure 4.4b shows the corresponding PCFG with additional information about use-definition and liveness analyses for the nodes significant to this case. In these nodes, UE stands for upwards exposed, $Kill$ stands for killed (defined), and LI stands for Live In.

Compiler analysis Algorithm 5 shows the code that, given a complete PCFG, checks the coherency of the data-sharing attributes of all tasks. For each task node, it first uses use-definition and liveness analyses to detect private variables that are defined within the task and the definition is dead, while the corresponding original variable is alive after the task synchronization. If a variable is reported to be in such a situation, no other checks are performed. Otherwise, the algorithm checks if firstprivate variables are upwards exposed and private variables are not upwards exposed.



(a) code snippet (b) simplified PCFG

Figure 4.4: Scenario illustrating different incoherences in the data-sharing attributes of a task.

Algorithm 5 High-level algorithm to detect incoherences in the data-sharing attributes of OpenMP tasks.

1: $incoherent_dead = incoherent_p = incoherent_fp = \varnothing$
2: **for each** $n_T \in N$ **do**
3:   **for each** $v \in \{private(n_T) \cup firstprivate(n_T)\}$ **do**
4:     **if** $\exists n_v : (n_T$ encloses $n_v \wedge v \in DEF(n_v))$
      $\wedge \nexists n : (n_T$ encloses $n \wedge n_v$ dominates n
      $\wedge v \in UE(n)) \wedge \exists n', e = (n_T, n', KIND) : v \in LI(n')$ **then**
5:       $incoherent_dead = incoherent_dead \cup v$
6:     **else**
7:       **if** $v \in private(n_T) \wedge v \in UE(n_T)$ **then**
8:         $incoherent_p = incoherent_p \cup v$
9:       **else if** $v \in firstprivate(n_T) \wedge v \notin UE(n_T)$ **then**
10:         $incoherent_fp = incoherent_fp \cup v$
11:       **end if**
12:     **end if**
13:   **end for**
14: **end for**
15: **return** $incoherent_dead$, $incoherent_p$, $incoherent_fp$

Applied to the example in Figure 4.4a: a) variable x' (task's local copy of variable x) is defined within the task but it is dead after that definition; the original x however is live after the task synchronization; b) variable y' (task's local copy of variable y) is private within the task and it is upwards exposed; and c) variable z' (task's local copy of variable z) is firstprivate within the task, and it is defined before being used.

Compiler solution The compiler suggests changes depending on each case:

- For *incoherently dead* variables like x it proposes to define the variable as shared.
- For *incoherently private* variables like y it proposes to define them as firstprivate.
- For *incoherently firstprivate* variables like z it proposes to define them as private.

4.3.5 Incoherent task dependences

Scenario OpenMP task dependences are used to impose an order in the execution of the tasks. This order has an impact on the performance because it reduces the parallelism of the tasks. There are three types of incoherences:

Incoherent Pointer: objects accessed via pointer must specify the dependence in the accessed storage instead of the pointer.

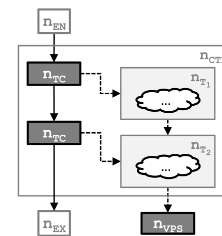
Incoherent In: input dependences should be upwards exposed and should not be defined.

Incoherent Out: output dependences should not be upwards exposed, and should be defined.

Example Figure 4.5a demonstrates that an over restrictive definition of the dependences of a task may cause the serialization of tasks that could run in parallel. Specifically, *Task2* defines an inout dependence on *A* whereas *A* is just read inside the task. Therefore, the dependence could be defined as input. As a consequence, *Task2* cannot start until *Task1* finishes its execution.

```
1 void foo(int* A, int* B, int* C, int i)
2 {
3   #pragma omp task depend(in:A) depend(out:B)     // Task1
4     B[i] += A[i];
5   #pragma omp task depend(inout:A) depend(out:C) // Task2
6     C[i] = A[i];
7 }
```

(a) code snippet



(b) simplified PCFG

Figure 4.5: Scenario illustrating the wrong specification of task dependences.

Compiler analysis Algorithm 6 shows the code that, given a complete PCFG, checks the coherency of the dependence clauses in all tasks. For each task node, the method gathers the input and output dependences. For each set, it uses use-definition analysis to check whether a dependence specified on a pointer variable should be instead specified on the pointed object. If not, it checks if input dependences are upwards exposed, and output dependences are defined.

Applied to the example in Figure 4.5a, all dependences should specify the pointed object (*A[i]*, *B[i]* and *B[i]*), instead of the pointer, which is never modified. In case the pointed objects were specified as dependences, then the algorithm will find that *Task2* has an incoherent output dependence on *A[i]*, which should be an input dependence.

Compiler solution To avoid this situation, the compiler suggests two actions in a specific order:

- First, define the dependences on the pointed objects in all four clauses.
- Then, after applying the first change, remove the output dependence on *A[i]*.

Algorithm 6 High-level algorithm to detect incoherences in the dependence clauses of OpenMP tasks.

1: $incoherent_ptr_in = incoherent_ptr_out = \varnothing$
2: $incoherent_in = incoherent_out = \varnothing$
3: **for each** $n_T \in N$ **do**
4:     **for each** $v \in in(n_T)$ **do**
5:         **if** v is pointer $\wedge$ $\exists v'$ subobject of $v : v' \in UE(n_T)$ **then**
6:             $incoherent_ptr_in = incoherent_ptr_in \cup v$
7:         **else if** $v \notin UE(n_T)$ **then**
8:             $incoherent_in = incoherent_in \cup v$
9:         **end if**
10:     **end for**
11:     **for each** $v \in out(n_T)$ **do**
12:         **if** $v \notin UE(n_T)$ **then**
13:             $incoherent_ptr_out = incoherent_ptr_out \cup v$
14:         **else if** $v \notin Kill(n_T)$ **then**
15:             $incoherent_out = incoherent_out \cup v$
16:         **end if**
17:     **end for**
18: **end for**
19: **return** $incoherent_in, incoherent_out, incoherent_ptr_in, incoherent_ptr_out$

4.4 Evaluation of the correctness tool

To evaluate the correctness framework we take two approaches: evaluate the usefulness based on the experience of novel programmers, and evaluate the accuracy compared to other similar tools. This section introduces the details of both evaluations.

4.4.1 Usefulness

In order to evaluate the usefulness of our tool we have used it in three courses of undergraduate students. The first was the "Course on programming models using OmpSs" [16] that took place in June 2014, in Bucaramanga, Colombia. This course had 21 participants, lasted for one week, and introduced basic and intermediate levels of OmpSs. The second and third courses were part of the "Parallelism" subject [153] of the Computer Science degree at the Technical University of Catalonia, Spain, which took place during May 2014 and October 2014. These courses had 23 participants (10 groups of 2 or 3 students) and 26 (13 groups of 2 students) respectively. Each course lasted for 3 weeks, and covered strategies for task decomposition and mechanisms for tasks synchronization. During the lectures, the students were asked to parallelize different algorithms using OpenMP and OmpSs tasking models, and analyze the performance and correctness of their implementations.

The students were provided with serial implementations or incomplete parallel versions of a series of benchmarks. They were given directions to perform the parallelization, and we applied quality checks on the results of the correctness tool at two different steps: a) before the tool was

given to the students, the expected mistakes were tested by myself, and b) during the lectures, the results were checked by the different professors of the lectures and myself. The assignments involve the next medium size programs:

- Compute the n_{th} number in the Fibonacci sequence: simple version and linked list version (appendix B.1.1).
- Compute the dot product of two equal-length arrays (appendix B.1.2).
- Compute the multiplication of two matrices (appendix B.1.3).
- Compute the number Pi with a Monte Carlo method (appendix B.1.4).
- Compute a solution for a random Sudoku puzzle (appendix B.1.5).

Figure 4.6 displays the results of this test. While all codes used in this evaluation involved data-sharing attributes, only one of them involved dependence clauses. This is the cause of having the most common mistakes related with the data-sharing attributes. The mistakes ordered by frequency are as follows:

1. Defining a variable that is never used, thus dead, due to using the firstprivate default data-sharing instead of explicitly defining it as shared.
2. Using a variable as firstprivate instead of private when its initial value is never read.
3. Having a race condition, either because a variable is not protected in an `atomic` or `critical` construct, or because the task is not properly synchronized.
4. Using an automatic storage variable in a task which is not synchronized in the scope of the variable.
5. Defining dependences on a pointer variable instead of on the pointed object.
6. Defining a variable as private when it should be firstprivate because it is upwards exposed.
7. Defining a variable as an input dependence when its value is never read.

The two last cases are not common because users have to explicitly determine the data-sharing attribute or the dependence clause, whereas for the other cases, the default data-sharing rules apply for the variables and usually programmers forget to explicitly change it.

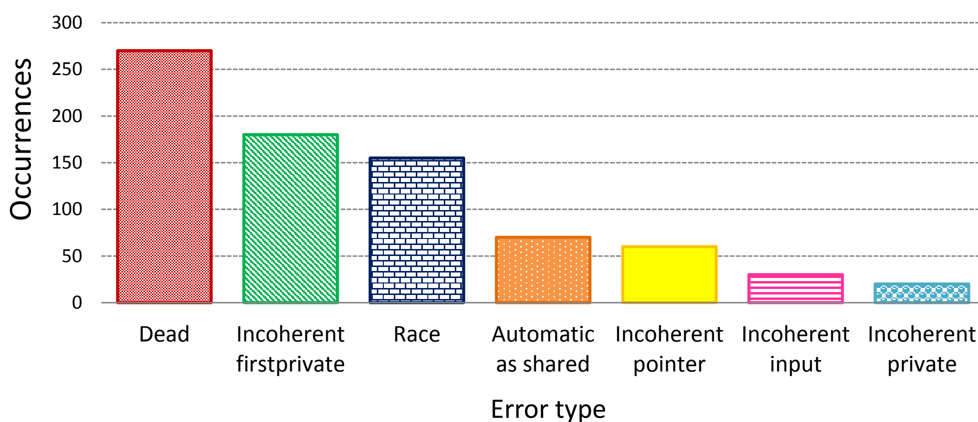


Figure 4.6: Occurrences of different correctness mistakes.

4.4.2 Comparison with other frameworks: Oracle Solaris Sudio 12.3

We also have compared our messages with those from the Oracle Solaris Studio 12.3 compiler [114]. The Studio compiler warns two different situations: parallelized loops with data dependences between different loop iterations, and problematic data-sharing attributes (e.g., declare as shared variables whose accesses in a parallel region might cause data race, and declare as private a variable whose value in a parallel region is used after the parallel region). The first situation is not useful for us to compare because it does not involve tasks, so we only analyze the second situation. Studio does not implement OpenMP 4.0 however, so the case study regarding dependence clauses (Section 4.3.5) cannot be compared.

We use the code snippets shown in each of the case studies presented in Section 4.3. The results are shown in Table 4.1 and analyzed as follows:

Case 1. Mercurium advices to synchronize the task instead of privatizing the variable because it is an array. Studio advises to firstprivatize the variable instead. We have used GCC to test the performance of the two versions with this simple code snippet. After 5 executions, the average time used in the version using firstprivate is 6.656ms, and the time used in the taskwait version is 2.709ms, which results in losing 4ms for this simple example.

Case 2. Studio shows a wrong message, since *x* is a global variable, meaning that it is accessible from every scope (unless it has been shadowed). In the example, the variable is around at any moment the task is executed. Additionally, the compiler does not warn about the real problem, i.e., the race condition. If we wrap the task and the call to *printf* in a `parallel` construct, then Studio is able to recognize the race. It remains unclear to us why the lack of a `parallel` construct results in a wrong message. Studio compiler is proprietary software and the only documentation is the Oracle web site, so we cannot analyze their algorithm.

Case 3. Oracle is not considering the possible loss of performance of firstprivatizing a variable which value is never read. We already proved in *case 1* that copying arrays may be unnecessarily expensive.

4.5 Impact

The works developed in the context of OpenMP correctness and programmability have had quite an influence in the community.

On one hand, Wang and Chen [160] drew from our algorithm for automatically determining the data-sharing clauses in OpenMP tasks to develop a different approach with the same goal. Their technique, however, is based on introducing taskwaits instead of analyzing the concurrent code, hence the performance can be severely affected. Additionally, Aldea [9] also considered our work on automatic scoping for OpenMP to develop an automatic generator of OpenMP directives and clauses needed to parallelize source code speculatively (thread-level speculation[2]).

[2] *Thread-level speculation* is a dynamic parallelization technique that depends on out-of-order execution to achieve speedup on multiprocessor CPUs [142].

Case	Oracle Solaris Studio 12.3	Mercurium

1
Fig. 4.1a
	`test.c, line 4: Warning: inappropriate scoping,`
	`variable 'a' may be scoped inappropriately as 'shared'`
	`* may not be around during the execution`
	`of task at line 4 is executed`
	`* consider 'firstprivate'`
		`test.c: 4: warning: OpenMP task defines as`
		`'shared' local data 'a' whose lifetime may`
		`have ended when the task is executed.`
		`Consider synchronizing the task before`
		`the local data is deallocated.`

2
Fig. 4.2a
	Without using a `parallel` construct:
	`test.c, line 4: Warning: inappropriate scoping,`
	`variable 'x' may be scoped inappropriately as 'shared'`
	`* may not be around during the execution`
	`of task at line 5 is executed`
	`* consider 'firstprivate'`
	Using a `parallel` construct:
	`"test.c", line 4: Warning: inappropriate scoping,`
	`variable 'x' may be scoped inappropriately as 'shared'`
	`* read at line 6 and write at line 5 may`
	`cause data race`
		`test.c: 4: warning: OpenMP task may have a`
		`race condition on 'x' because other threads`
		`access concurrently to the same data.`
		`Consider synchronizing all concurrent`
		`accesses or privatizing the variable.`

3
Fig. 4.4a
	`"test.c", line 4: Warning: inappropriate scoping,`
	`variable 'x' may be scoped inappropriately`
	`as 'firstprivate'`
	`* write at line 7 may be used outside:`
	`read at line 10`
	
	`"test.c", line 4: Warning: inappropriate scoping,`
	`variable 'y' may be scoped inappropriately`
	`as 'private'`
	`* read at line 6 may be undefined`
	`* consider 'firstprivate'`
		`test.c:4: omp-warning: Variable 'x' is`
		`firstprivate, therefore, updates on this`
		`variable will not be visible after the task.`
		`Consider defining it as shared.`
		
		`test.c:4: omp-warning: Variable 'y' is private`
		`in the task, but its input value would have`
		`been used in a serial execution.`
		`Consider defining it as firstprivate instead,`
		`to capture the initial value.`
		
		`test.c:4: omp-warning: Variable 'z' is`
		`firstprivate in the task, but its input value`
		`is never read.`
		`Consider defining it as private instead`

Table 4.1: Oracle Solaris Studio and Mercurium messages for different correctness situations.

On the other hand, Papakonstantinou et al. [115] distinguishes our work on the automatic definition of task dependence clauses, together with SCOOP [165], as the only off-line tools for task dependence detection. They study the possibilities of a combined off- and on-line tool and and compare their results with those of Mercurium/Nanos.

4.6 Conclusion

Using OpenMP to easily parallelize applications is attractive because of its programmability. Nonetheless, knowing the internals of the language may not be as easy as expected. Furthermore, debugging parallel programs to find errors at run-time can be arduous. In this context, the compiler is a key tool to help programmers finding correctness errors.

The presented algorithms are based on classic techniques of control and data-flow analysis. They include OpenMP support to detect correctness mistakes related with synchronizations, data-sharing attributes and dependence clauses associated with tasks. We classify these mistakes in different case studies and propose solutions to fix them. We also implement all the algorithms in the Mercurium source-to-source compiler, which we use to test the usefulness of the proposal with a number of end-users and benchmarks. On one hand we test our tool with several students, gathering logs of their executions to study the more common errors. On the other hand, we compare our results with those of Oracle Solaris Studio which, to the best of our knowledge, is the only compiler implementing such a correctness checking feature.

Based on our tests, OpenMP beginner programmers often make mistakes related to the data-sharing attributes and the dependence clauses. These mistakes are mainly related with the default data-sharing attributes, e.g., programmers forget to explicitly determine as shared variables

which are firstprivate by default, or they forgot the explicitly determine as private variables which are firstprivate by default. The first case leads to a wrong result of the program, whereas the second leads to a loss of performance due to an unnecessary copy. The other most common mistake is to define a program with race conditions as a result of a wrong synchronization of either the tasks or the access to the variables. This mistake leads to non-deterministic results.

According to our comparison with Oracle Solaris Studio, the algorithms implemented in Mercurium cover more cases and propose more accurate hints. While Mercurium addresses both correctness and performance issues, Studio only tackles correctness. Messages related with the variables involved in a race condition are more accurate in the Studio compiler though in the sense that they point out which accesses are in a race.

Even experienced programmers can make mistakes very hard to find at run-time, especially in large codes. This is why a compile-time tool providing correctness tips is always useful and effortless from the programmer point of view.

5

A Static Task Dependency Graph for OpenMP

OpenMP is increasingly being adopted by modern many-core embedded processors to exploit their parallel computation capabilities. Unfortunately, current OpenMP runtime libraries are not suitable for processors relying on small and fast on-chip memories, due to its memory consumption. In this part of the thesis we present a complete tool-chain that enables the execution of codes based on the OpenMP tasking model on such systems. The tool-chain is based on a compiler transformation that is able to statically generate a TDG, and the runtime support to execute this TDG instead of the regular mechanism for dynamic checking. The reduction in memory consumption is accomplished as a result of the efficiency of the mechanism used to store and access the TDG.

5.1 Applicability

Although OpenMP was originally focused on massively data-parallel loop-intensive applications, the latest specifications have evolved to support dynamic and irregular parallelism, as well as heterogeneity. By virtue of these extension, the language has gained much attention in the real-time embedded domain [32, 159]. Furthermore, real-time applications are usually modeled as a Directed Acyclic Graph (DAG) to analyze its timing and functional properties, and the OpenMP tasking model can be represented as a TDG [155], a type of DAG. For that reason, the tasking model is suitable to exploit the capabilities of current many-core embedded platforms (e.g., Kalray MPPA [41], STM P2012 [28], TI Keystone II [144]), and thus deliver the level of performance required to face current and future challenges of embedded systems.

Current implementations of OpenMP (e.g., libgomp [60], Nanos++[15]) generate the TDG at run-time for two reasons: 1) the TDG depends on the tasks that are instantiated, which is determined by the CFG, and 2) the addresses of the data elements upon which dependences are built are known at run-time. Consequently, large data structures are required to manage the tasking model, as shown in Table 5.1 (The size of the structure that holds a task depends on the variables used within the task and the dependencies the task may have with other tasks. Nanox uses significantly more memory because it supports a more complex model to that of OpenMP, as

Runtime library	**libgomp**		**Nanox++**
	GCC 5.4	**GCC 7.1**	
Size(Bytes)	176	208	1056

Table 5.1: Minimum memory (in Bytes) used to store an OpenMP task in different runtimes.

introduced in Section 2.1.2). Modern many-core embedded designs, however, rely on computing platforms with small on-chip memories that are accessible by a limited number of cores (usually organized in clusters), making these runtimes unsuitable.

As an illustration, we examine the memory consumption of the libgomp runtime library. In this implementation, when a new task is created, its `in` and `out` dependences are matched against those of the existing tasks. To do so, each task region maintains a *hash table* that stores the memory address of each data element defined in the `out` and `inout` clauses, and the list of tasks associated with the task it represents. Each position of the hash table is linked to those tasks depending on the object it represents. The runtime can quickly identify which successors may be ready for execution when the task completes by accessing its hash table. This table is cleared when a task reaches a `taskwait` or a `barrier`, when all pending tasks must be resolved. Removing the information of a single task at completion may turn out to be very costly, because dependent tasks are tracked in multiple linked lists. As a result, memory consumption may significantly increase as the number of concurrently instantiated tasks increases.

We prove that storing a complete statically generated TDG can result in a huge reduction of the memory used at run-time. Although this idea may seem counter-intuitive, the data structures needed to store a static TDG are much lighter than those necessary to dynamically build the TDG. Moreover, statically deriving the TDG provides an extra benefit: it allows applying real-time DAG scheduling models [24], from which timing guarantees can be derived [96, 139].

5.2 Related work

Sarkar et al. [136] presented a framework for partitioning and scheduling tasks at compile-time balancing tasks granularity, and thus overhead and parallelism. Vijaykumar et al. [157] proposed a set of heuristics to generate a TDG for massively data-parallel applications based on the CFG and use-definition analysis, aiming at reducing communication and synchronization overheads. Yet none of these methodologies are able to create at compile-time a TDG for complex and irregular algorithms.

Pugh et al. [30] proposed a new technique to detect dependencies at compile-time and map HPC kernels into cluster nodes. The disadvantage is that it is expensive and introduces overhead to the compiler while, in our proposal, much simpler dependency analyses are enough to check tasks parallelism.

Tzenakis et al. [152] implemented task instantiation, dependence analysis and scheduling techniques, and proved their efficiency over runtimes such as SMPSs [13]. However, this method has runtime overhead and requires heavy data-structures for dynamic dependency checking.

Finally, Arandi et al. [10] presented a hybrid approach to try to get the best of both static and dynamic methods, but the technique still introduces too much overhead, as the authors admit.

Furthermore, Liu et al. [89] proposed an OpenMP runtime for multi-core platforms with limited memory resources. However, this runtime only implements OpenMP 2.5, in which the tasking model is not supported.

5.3 Compiler analysis

Based on the analyses introduced in Chapter 3, we have developed a new phase in the Mercurium compiler (see Section 2.2.1 for further details) that generates a TDG out of a source code based on the OpenMP tasking model. The following sections explain the analyses, transformations and implementation decisions that apply to this feature. To illustrate all stages, we use the code shown in Listing 5.1, that performs a computation over a matrix using a wavefront strategy, meaning that the processing of block (i, j) depends on blocks $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$. These dependences are expressed in the dependence clauses of each task.

```
1 #define N 2
2 #define BS 16
3
4 extern void compute_block(int i, int j);
5
6 void wavefront(long m[N][N][BS][BS])
7 {
8    #pragma omp parallel
9    #pragma omp single nowait
10   {
11      for (int i=0; i<=N; i++) {
12         for (int j=0; j<=N; j++) {
13            if (i==0 && j==0)
14            {  // Initial block
15               #pragma omp task depend(inout:m[i][j])                     // T1
16               compute_block(i, j);  // Task region T1
17            }
18            else if (i == 0)
19            {  // Blocks in the upper edge
20               #pragma omp task depend(in:m[i][j-1], inout:m[i][j])       // T2
21               compute_block(i, j);  // Task region T2
22            }
23            else if (j == 0)
24            {  // Blocks in the left edge
25               #pragma omp task depend(in:m[i-1][j], inout:m[i][j])       // T3
26               compute_block(i, j);  // Task region T3
27            }
28            else
29            {  // Internal blocks
30               #pragma omp task depend(in:m[i-1][j], in:m[i][j-1], \\     // T4
31                                       in:m[i-1][j-1], inout:m[i][j])
32               compute_block(i, j); // Task region T4
33            }
34         }
35      }
36   }
37 }
```

Listing 5.1: OpenMP tasks example traversing a matrix with a wavefront strategy.

5.3.1 Control and data flow analysis

The generation of a TDG requires the identification of the control flow statements[1] that determine if a task is instantiated, and the conditions to fulfill for two tasks to be dependent. To that end, we generate a PCFG (see Section 3.2.1 for further details) where synchronization edges are augmented with predicates defining the condition to be fulfilled for the edge to exist. Furthermore, the compiler evaluates the iteration statements to discover the induction variables and their evolution over the iterations, and therefore the iteration space. This information is sufficient to generate a *flow TDG* (fTDG). This is a TDG with one node per each `task`, `taskwait` or `barrier` node found in the PCFG. Additionally, each node in the fTDG is augmented with information about the control flow structures surrounding it. Finally, nodes are connected according to the synchronizations they may cause (tasks are connected between them based on the dependence clauses, and tasks are connected to taskwait and barrier nodes according to the specification). Hence, a *flow Task Dependency Graph*, $f\mathcal{TDG}$, is a tuple

$$f\mathcal{TDG} = \langle N, E, C \rangle$$

where:

- $N = \{V \times T_N\}$ is the set of nodes with its type $T_N = \{Task, Taskwait, Barrier\}$.
- $E = \{N \times N \times P\}$ is the set of possible synchronization edges with the predicate P that must fulfill for the edge to exist.
- $C = N \times \{F\}$ is the set of control flow statements involved in the instantiation of any node (task, taskwait or barrier), $n \in N$, where $F = S \times \{T_F\}$, being S the condition to instantiate the node and $T_F = \{Loop, IfElse, Switch\}$, the type of the structure.

Figure 5.1 shows the fTDG of the OpenMP program in Listing 5.1. It includes:

- the set of nodes $N = \{T1, T2, T3, T4, B\}$ from lines 15, 20, 25, 30 and 36. The four first nodes with type $T_N = Task$, and the last, corresponding to the implicit barrier at the end of the parallel region, with type $T_N = Barrier$.
- the control flow statements `for` and `if` $f_i \in F$ from lines 11, 12, 13, 18, 23 and 28, attached to the corresponding tasks in N. These include information about: a) the induction variables of each loop i, j, both with lower bound $lb = 0$, upper bound $ub = 2$ and stride $str = 1$ (dashed-line boxes), b) the conditions of the selection statements enclosing each task (solid-line boxes), and c) the ranges of the variables in those conditions, e.g., $T3$ is instantiated if $i = 1$ or 2 and $j = 0$.
- the predicates $p \in P$ associated with the synchronization edges in E, where the left hand side of the equality corresponds to the value of the variable at the point in time the source task is instantiated, and the right side corresponds to the value when the target task is instantiated. For example, the predicate $p1 = ((i_s == i_t \,\|\, i_s == i_t - 1) \,\&\&\, j_s == j_t)$ of the edge between T_1 and T_3, evaluates to *true*, meaning that the edge exists, when $i_s = 0, j_s = 0$ for T_1 and $i_t = 1, j_t = 0$ for T_3. For simplicity, the fTDG only shows the dependencies that are actually expanded in the next stage (Section 5.3.2). The actual fTDG has edges between any possible pair of tasks because they all have `inout` dependences on the element $m[i][j]$.

[1] In a C/C++ program, *control flow* statements are selection statements, iteration statements, and jump statements.

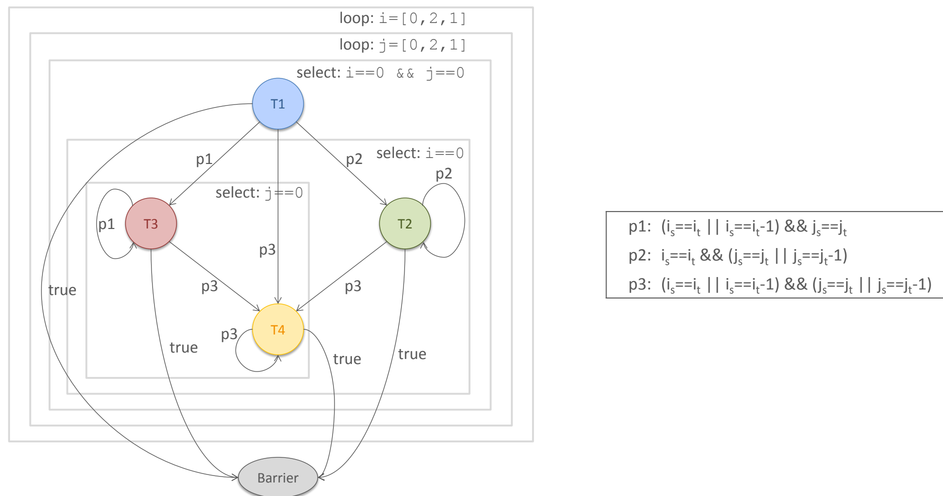


Figure 5.1: fTDG of the OpenMP program in Listing 5.1.

5.3.2 Task expansion

The fTDG contains the information necessary to expand a complete TDG representing all tasks that will actually be executed, and the synchronizations existing among them. The expansion is performed from outer to inner nodes, and the values of the constants and variables involved in the expansion (and resolved in the previous expansion stage) are propagated to inner nodes.

A crucial aspect is to match the tasks expanded at compile-time with the tasks instantiated at run-time. With that goal, the compiler inserts two identifiers: 1) a unique identifier for each task construct, tc_{id} (we use consecutive values starting by one), and 2) a unique identifier for each loop expansion step of each loop, l_{id}. The equation used to determine the identifier of a given task instance, t_{id}, in both the compiler and the runtime, is the following:

$$t_{id} = tc_{id} + T \times ((((((l_1 \cdot I) + l_2) \cdot I) + ...) + l_L) \cdot I)$$

where,

- tc_{id} is the identifier of the task construct that is being expanded.
- T is the total number of task constructs in the source code.
- l_n is the unique identifier of loop at nesting level n ($n \in [1, L]$, where L is the number of loops involved in the expansion of t_{id}).
- I is the maximum number of iterations of any loop used during expansion.

Since a task construct can generate multiple task instances, we use loop properties (l_n, L and I) to guarantee that each task instance identifier is unique. As a result, task instances from different loop iterations will result in different t_{id} because every nesting level l_n is multiplied by the maximum number of iterations I.

Each time a task is expanded, the possible dependences with previous tasks are resolved by evaluating the predicates of the fTDG (all values involved in the predicates must be known at this point). Additionally, all transitive dependences are removed because they are redundant.

Figure 5.2 shows the expanded TDG of the program in Listing 5.1. It contains all task instances and all dependences that could exist at run-time (redundant edges such as that between task 100 and task 148 do not appear). Each instance contains the task id t_{id}.

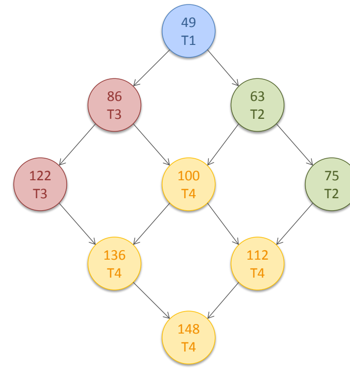


Figure 5.2: TDG of the OpenMP program in Listing 5.1.

As an illustration of the expansion of a TDG, consider task T4 with identifier 136 from Figure 5.2, which corresponds to the computation of the matrix block $m[2,1]$. The identifier t_{id} is computed as follows: 1) tc_{id} = 4, because T4 is the fourth task in sequential order found while traversing the source code; 2) $T = 4$ because there are four task constructs in the source code; 3) $L_{T4} = 2$ because there are two loops enclosing T4; 4) $I = 3$ because three is the maximum number of iterations in any of the two considered loops; 5) $l_1 = 3$ because the instance is created in the third iteration of the first loop (outer loop), and $l_2 = 2$ because the instance is created in the second iteration of the second loop (inner loop). Putting all together: $T4_{id} = 4 + 4 * (((3 * 3) + 2) * 3) = 136$. Then, the dependences with previous tasks are resolved by evaluating the predicates. We perform a bottom-up breadth-first traversal, and we stop traversing when we find a node whose predicate evaluates to true. This way, we avoid creating transitive edges. Predicate $p3$ is the only one to be evaluated for task 136. For task instance 122, the predicate evaluates as $(2 == 2 \,||2 == 2 - 1)$ && $(0 == 1 \,||\, 0 == 1 - 1) = TRUE$, and for task instance 100, the predicate evaluates as $(1 == 2 \,||\, 1 == 2 - 1)$ && $(1 == 1 \,||\, 1 == 1 - 1) = TRUE$.

5.3.3 Missing information when deriving the TDG

The compiler is able to fully expand the TDG when all variables involved in the control flow structures that are to be expanded are known at compile-time. This may not be possible when using pointers or complex array indexes. However, missing information cannot prevent the application from validating, and we propose the following solutions for each case:

- If a selection statement cannot be evaluated, all the possible paths are expanded.Two situations are equivalent at run-time: 1) a predecessor task never existed because the associated condition evaluates to false, and 2) a predecessor task has already been executed. As a result, it is not wrong to define a dependency between two tasks if one of them eventually does not exist.
- If a loop cannot be expanded because its boundaries are unknown, parallelism across iterations can be disabled by inserting a `barrier` (if there is nested parallelism) or a `taskwait` (otherwise) at the end of the loop.
- If the predicate of a dependence cannot be evaluated, we assume it evaluates to true and hence we keep the dependence. This forces the tasks to be sequentialized.

Some of these solutions come at a cost. On one hand, if many selection statements cannot be evaluated, the TDG may increase considerably. This has no impact in the performance, but the

amount of resources required by the application to run may increase considerably. On the other hand, when a loop cannot be evaluated, the performance may be affected because the solution proposed results in the sequentialization of that particular loop. The impact in performance for sequentialized loops depends on the weight these loops have in the total execution time of the application. Finally, in the worst-case scenario, where no information can be derived at compile-time, the TDG corresponds to the sequential execution of the program.

5.3.4 Communication with the runtime

The runtime must be able to compute the same identifier for a given task instance as the compiler does. For that reason, the compiler introduces the following modifications in the generated code:

- The identifier of the task construct is introduced as a new clause of the form `task_id(int)`.
- The total number of task constructs, T, and the maximum number of iterations of any loop, I, are defined in an intermediate file generated by the compiler and linked with the final binary.
- In order to obtain the same l_n at compile-time and at run-time, the compiler introduces a *loop stack* for each loop, and *push* and *pop* before the loop begins and after it ends, respectively. At every loop iteration the top of the stack is increased by 1. These operations are only included in those loops containing tasks, and the overhead introduced by these operations is negligible compared to the time expended in the creation and destruction of the tasks.

5.3.5 Complexity

The complexity of the compiler is defined by the complexity of the two phases of the process that derives the TDG: 1) the generation of the PCFG and the analysis of induction variables, and 2) the expansion of the TDG.

The complexity of the control-flow and data-flow analysis stage is dominated by the complexity of the PCFG, which is related to the number of control flow statements present in the source code, in which Cyclomatic Complexity [95] metric is usually used.

The complexity of the task expansion stage is dominated by the computation of the dependences among tasks, which is performed using a Cartesian product: the input dependence of a task can be generated by any of the previously created task instances. As a result, the complexity is quadratic on the number of instantiated tasks.

5.4 Runtime support [2]

The runtime uses a *sparse matrix* to store the TDG, and schedules tasks while honoring their dependences based on this TDG instead of using the dependence clauses. Figure 5.3 shows the sparse matrix implementation of the TDG presented in Figure 5.2. Each entry of the matrix contains an unique task identifier t_{id}, and stores in separate arrays the tasks it depends on (input

[2]The runtime support has been developed by Vargas et.al [156], but we find interesting to explain some details here to understand the whole tool-chain.

dependences), and the tasks depending on it (output dependences). Moreover, the sparse matrix is sorted using the t_{id}, so a dichotomic search can be applied.

Additionally, each task instance entry t_i in the sparse matrix has an associated counter (not shown in the figure) indicating the state as the number of tasks of which the entry still depends on (these tasks have been created and not completed yet). The counter is -1 if the task has not been instantiated or has finished; it is 0 if the task is ready to run; and it is > 0 if the task is waiting its input tasks to finish.

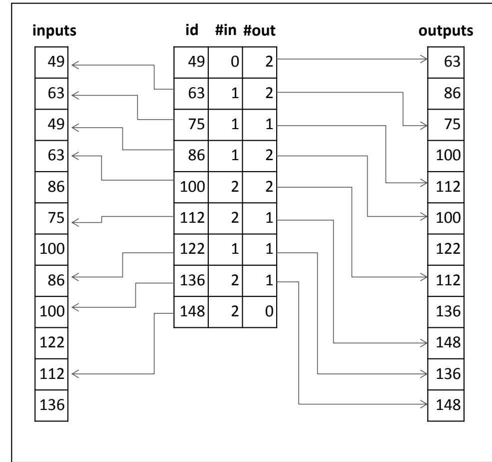


Figure 5.3: Hash table that stores the TDG depicted in Figure 5.2, corresponding to the OpenMP program in Listing 5.1.

The runtime task scheduler works as follows: when a new task is created, the runtime checks the state of its *input* tasks. If all counters are -1, then the task is ready to execute, and its counter is set to 0; otherwise, the counter of the new task is initialized with the number of input tasks with a state ≥ 0. When a task finishes, it decrements by 1 the counters of all its output tasks whose counter is > 0.

5.5 Evaluation

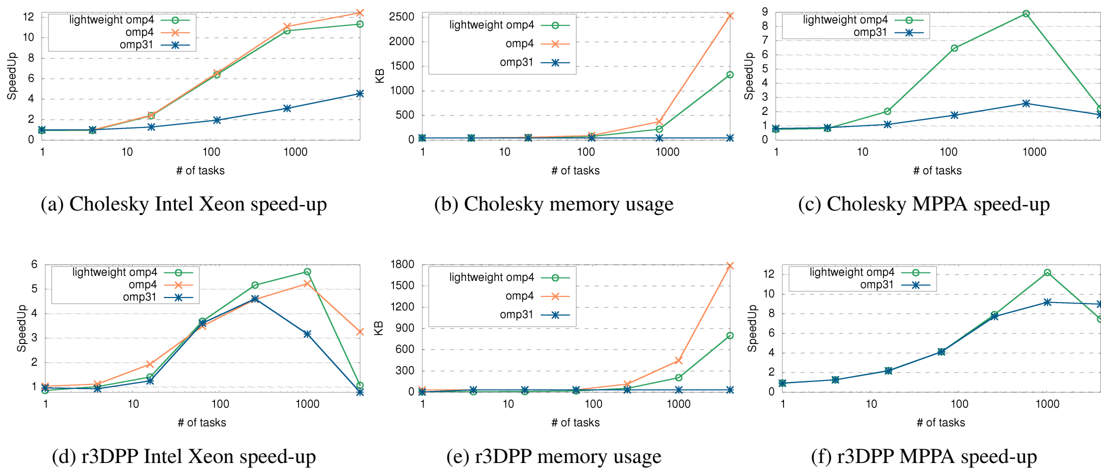
The evaluation of this work has been performed in the frame of the P-Socrates European project [121], where the Kalray MPPA processor was used (see Section 2.3.2 for more details). The board used supports GCC 4.7.2.

5.5.1 Experimental setup

OpenMP framework. All the analysis and transformations presented in Section 5.3 have been developed in the Mercurium compiler (see Section 2.2.1 for more details). The runtime support has been developed in libgomp from GCC 4.7.2 (see Section 2.2.2 for more details). This version of GCC implements OpenMP 3.1, thus dependence clauses are not supported. We also consider the libgomp from GCC 4.9.2, which implements OpenMP 4, for comparison purposes.

Applications. For the evaluation we consider applications from two different domains:

- From the HPC domain, we consider a *Cholesky factorization* [25], useful for efficient linear equation solvers and Monte Carlo simulations. Cholesky can also be used to accelerate *Kalman filter*, implemented in autonomous vehicle navigation systems to detect pedestrians and bicyclists positions [84].
- From the embedded domain, we consider an application resembling the *3D path planning* [127] (r3DPP), used for airborne collision avoidance.



(a) Cholesky Intel Xeon speed-up (b) Cholesky memory usage (c) Cholesky MPPA speed-up

(d) r3DPP Intel Xeon speed-up (e) r3DPP memory usage (f) r3DPP MPPA speed-up

Figure 5.4: Performance speed-up and memory usage (in KB) of Cholesky and r3DPP applications running with *lightweight omp4, omp4* and *omp 3.1*, and varying the number of tasks.

We have implemented two versions of both applications: one using task dependence clauses, and the other using the `taskwait` construct as synchronization method.

Platform setup. We run our experiments in two different systems: 1) a computing node of the MareNostrum III supercomputer, which consists of two Intel Xeon CPU E5-2670 processors, featuring 8 cores each, with 20 MB L3, and 2) the MPPA processor featuring 256 cores organized in 16 clusters of 16 cores each, and 2 MB of private on-chip memory per cluster. The former executes a complete Linux system, in which OpenMP 3.1 and OpenMP 4.0 are supported; the latter, only supports OpenMP 3.1. (Details of these platforms are provided in Section 2.3.)

5.5.2 Performance speed-up and memory usage

Figure 5.4a and Figure 5.4d show the performance speed-up achieved by Cholesky and r3DPP respectively in the Intel Xeon processor, when varying the number of instantiated tasks, ranging from 1 to 5984 and 4096 respectively. We consider three libgomp runtimes: OpenMP 4.0, OpenMP 3.1 and OpenMP 3.1 augmented with our dependency checker (labeled as *omp4*, *omp 3.1* and *lightweight omp4* respectively).

The performance has been computed with the average of 100 executions. Similarly, Figure 5.4b and Figure 5.4e show the heap memory usage (in KB) of the three OpenMP runtimes when executing Cholesky and r3DPP respectively in the Intel Xeon processor and varying the number of instantiated tasks as well. The memory usage has been extracted using *Valgrind Massif* [104] tool, which allows profiling the heap memory consumed by the runtime in which the TDG structure is maintained.

We observe that both performance and memory usage depend on the number of instantiated tasks: the higher the number of instances, the better the performance, as the chances of parallelism increase. When the number of tasks is too high, however, the overhead introduced by the runtime, the small workload of each task and the NUMA effect slows-down the performance.

Cholesky	**Tasks**	4	20	120	816	5984
	KB	0.11	0.59	3.80	27.09	204.19
r3DPP	**Tasks**	16	64	256	1024	4096
	KB	0.47	1.94	7.88	31.75	127.5

Table 5.2: Memory usage of the sparse matrix (in KB), varying the number of tasks instantiated.

Our *lightweight omp4* obtains the same speed-ups as the *omp4* implementation, and outperforms *omp 3.1*. However, in case of *omp4*, the memory usage rapidly increases, requiring much more memory than our runtime.

The parallelization opportunities brought by the `depend` clause make the performance of Cholesky (Figure 5.4a) to increase significantly compared to the OpenMP 3.1 model, with a speed-up increment from 4x to 12x when instantiating 5984 tasks. At this point, *omp4* consumes 2.5MB while our *lightweight omp4* requires less than 1.3MB. The memory consumed by *omp3.1* is less than 100KB (Figure 5.4b). In fact, the *omp3.1* memory consumption is similar for all the applications because no structure for dependencies management is needed.

For the r3DPP (Figure 5.4d), the tasking model achieves a performance speed-up of 5.2x and 5.8x with *omp4* and *lightweight omp4* respectively, when instantiating 1024 tasks. At this point, *omp4* consumes 400 KB in front of the 200 KB consumed by *lightweight omp4* (Figure 5.4e). *omp31* achieves a maximum performance of 4.5x when 256 tasks are instantiated. When the number of task instances increases to 4096, all runtimes suffer a significant performance degradation because the number of instantiated tasks is too high compared to the workload computed by each task. The *lightweight omp4* suffers a higher performance penalization due to the dichotomic search.

Taking a deeper look into the memory consumption reported in Figures 5.4b and 5.4e, we show in Table 5.2 the size of the sparse matrix data structure implementing the TDG of each application when varying the number of instantiated tasks.

Finally, to evaluate the benefit of OpenMP 4.0 on a memory constrained many-core architecture, we run our lightweight runtime on the MPPA processor. Figures 5.4c and 5.4f show the performance speed-up of Cholesky and r3DPP executed in one MPPA cluster, considering the *lightweight omp4* and *omp31* runtimes, and varying the number of tasks (*omp4* experiments are not provided because MPPA does not support it). Memory consumption is the same as the one shown in Figures 5.4b and 5.4e. r3DPP increases the performance speed-up from 9x to 12x when using our *lightweight omp4*, and only consumes 200 KB. Cholesky presents a significant speed-up increment when instantiating 816 tasks, from 2.5x to 9x, consuming only 220 KB.

5.5.3 Impact of missing information when expanding the TDG

The impact of missing information when expanding the TDG may vary depending on the amount of unknown data. For example, if a task is in a selection statement and we are not able to evaluate it, then the less time the task is actually instantiated, the bigger the noise introduced in the TDG in order to keep correctness.

In order to properly measure the impact of missing information in terms of TDG size we evaluate the case in which the compiler cannot obtain all information from r3DPP. First, we identify those if-else statements with the highest and lowest impact (in each scenario the compiler is unable to determine 25% of the conditions). Assuming that 1024 tasks are instantiated (peak performance), the scenario with the highest impact increases the TDG by 126.67% (71.97 KBs); the scenario with the lowest impact increases the TDG by only 7.77% (34.22 KBs).

5.6 Impact

The static generation of a TDG has had an important impact in the field of real-time and schedulability analysis. Both Melani et al. [96, 138] and Serrano et al. [138] have used the static TDGs generated by Mercurium to argue about different scheduling techniques and response-time analysis for OpenMP. This is a very important milestone because the vast majority of literature about scheduling is based on synthetic graphs, while those works are based on real applications with real graphs. Furthermore, Guan et al. [145, 161] substantiates its work on scheduling OpenMP tied tasks in our method.

At the same time, the HPC domain also benefits from this work in three different directions:

- The use of the TDG for the study and implementation of a data-flow runtime [54] for the Xilinx All Programmable SoC [92]. Without the complete TDG expanded at compile time it would not be possible to generate code for such a runtime, because each task must know the tasks that will depend on it prior to generating code.
- The development of static scheduling techniques for OmpSs clusters [31]. In this case, the use of a TDG at compile time can be used to predict better schedulers for FPGAs based on the cost of the tasks, the cost of the communication, the data locality, etc.
- The exploitation of different degrees of granularity in parallel codes. In this regard, a statically generated TDG could be used for enhancing the performance of the smallest kernels by preallocating data at compile time, and implementing prefetching policies.

5.7 Conclusions

Memory consumption is not a problem in HPC systems, in which large amounts of memory are available. However, this is not the case in the newest many-core embedded architectures, as the MPPA processor, integrating 16 clusters of 16-cores each, with a 2 MB on-chip private memory per cluster. Despite the overall size of the MPPA memory is 32 MB, clusters only have access to their private memory. The rest of memory is accessible through DMA operations (with a significant performance penalization), and so the complete program (including the OpenMP runtime library) must reside within the private memory. Therefore, it is of paramount importance that the memory consumed by the runtime is reduced to the bare minimum.

Considering its characteristics, the MPPA (like other many-core embedded processors) only supports older OpenMP specifications (version 3.1) with no task dependence features. There

is therefore a need to implement memory efficient OpenMP 4.0 runtimes to fully exploit the performance opportunities of these platforms.

Our proposal to statically build the TDG allows the creation of a more lightweight OpenMP 4.0 runtime that reduces the memory consumed by the tasking data structures, while maintaining the same performance of current implementations. This enables the execution of OpenMP 4.0 programs in memory-constrained environments such as the MPPA.

Regarding the limitations during task expansion, we prove that the tool-chain is always able to run valid applications, although performance may be compromised. Nonetheless, embedded applications frequently allow deriving all the required information to complete the TDG expansion, as it is required for timing analysis [163] as well.

6

Towards a Functional Safe OpenMP

Critical real-time embedded systems can benefit from the flexibility delivered by OpenMP. Yet, the impact of the language in such a domain is very limited. The reason is that critical real-time systems require *functional safety* guarantees, imposing the system to operate correctly in response to its inputs from both functional and timing perspectives. Functional safety is verified by means of safety standards as the ISO26262 [72] for automotive, the DO178C [43] for avionics or the IEC61508 [71] for industry. The use of reliability and resiliency mechanisms allow guaranteeing the correct operation of the (parallel) execution. Moreover, the complete system stack must be guaranteed, from the processor architectural perspective (e.g., multi-core processor designs ARM Cortex-A57 [11] and Infineon AURIX [68] are safety compliant) to the operating system (e.g., PikeOS [75], VxWorks [126] and Erika Enterprise [141] are safety compliant).

In this chapter we address the application of OpenMP to critical real-time systems, from the specification to the implementation. The contributions of this chapter (organized in the next sections) are as follows:

1. Analysis of the specification of OpenMP to identify the features that may entail a hazard regarding functional safety, and solution proposed for each threat.
2. Study of the application of OpenMP to Ada, a language widely used in critical real-time systems by virtue of its specification. This entails the analysis of the Ada and the OpenMP execution and memory models and the study of its compatibility. This work also includes the empirical research of the interoperability of the two runtimes.
3. Development of compiler correctness techniques specific for mixed Ada/OpenMP applications to ensure safety in such programs.

6.1 Is OpenMP a suitable candidate for critical real-time systems?

The current OpenMP specification lacks the reliability and resiliency mechanisms necessary in safety-critical systems, at both compiler and runtime levels, to meet its safety requirements. However, OpenMP is still a suitable candidate to exploit parallelism in such systems by virtue of many factors (already introduced across the chapters of this thesis):

1. During 30 years, the OpenMP community has built a solid and productive model gathering the virtues of many other languages:

   (a) It delivers levels of performance comparable to highly tunable models such as TBB [76], CUDA [83], OpenCL [140] and MPI [79].
   (b) It has different advantages over low level libraries such as Pthreads [106]: i) it offers robustness without sacrificing performance [81], and ii) it does not lock the software to a specific number of threads.
   (c) The code can be compiled as a single-threaded application just disabling support for OpenMP, thus easing debugging.
2. The extensions included in the latest specification meet the characteristics of current heterogeneous architectures:
   (a) The coupling of a main host processor to one or more accelerators, where highly-parallel code kernels can be offloaded for improved performance/power consumption.
   (b) The capability of expressing fine-grained, both structured and unstructured, and highly-dynamic task parallelism.
   (c) The model is widely implemented by several chip (e.g., TI Keystone [144], Kalray MPPA [42], STM P2012 [28]) and compiler vendors (e.g., GNU [60], Intel [69], and IBM [66]), thus easing portability.
3. Although lacking resiliency and reliability mechanisms in its current specification, many works, including ours, pursue the introduction of such concepts:
   (a) Several compiler and runtime analysis techniques [27, 47, 87, 93, 103] have been developed over the years specifically for OpenMP shortening the distance towards a more reliable language.
   (b) Many algorithms have been presented to enhance the programmability [129, 131] and provide correctness information to the user [86, 133].
   (c) There are attempts to introduce resiliency mechanisms [46, 164] in the specification, and the last specification already included one of them, the cancellation.
   (d) Current works have analyzed the response time of both the thread-centric [53, 82] and the task-centric model [139, 155] to be time predictable.

All these reasons have inspired us to study the fitting of OpenMP in the domain of safety-critical applications, which we dissect in the next sections.

6.2 The OpenMP specification from a safety-critical perspective

This section discusses the OpenMP specification with the aim of: a) detecting those features that can be a hazard regarding functional safety, and b) proposing solutions to avoid the hazard at design-time, compile-time or run-time, depending on the case.

6.2.1 Related work

Parallel heterogeneous embedded architectures certainly require the use of parallel programming models to provide high throughput, low latency and energy-efficient solutions. Efforts to introduce OpenMP in such environments [38, 94] reveal that OpenMP runtimes can efficiently be aware of

the heterogeneity and the memory hierarchy to deliver good performance. However, all works that intend to introduce OpenMP in the embedded domain conclude that, although the language is very useful in such environments, some extensions with real-time processing and power-awareness functionalities are needed [62].

Critical real-time embedded systems add additional, more restrictive, constraints to those of the embedded domain. Concretely, timing guarantees and functional safety. Regarding the former, significant attempts to analyze the time predictability properties of OpenMP [139, 155] as well as deriving response time analysis for both work-conserving dynamic and purely static schedulers [82, 96, 138], confirm the OpenMP tasking model as a perfectly suitable parallel pattern for safety-critical environments. In this sense, the suitability of the thread-centric model still remains unproved. Furthermore, situations such as starvation when a barrier construct is found shall be addressed. Regarding the latter, functional safety, different works have tried to study, classify and solve mistakes commonly appearing in OpenMP applications [102, 146]. These works are very useful mostly for inexperienced programmers in order to avoid errors. Beyond the theoretical approaches, many articles propose different techniques tackling correctness in general, and OpenMP correctness in particular. Section 6.2 introduces several techniques for detecting specific errors in concurrent programs (i.e., race conditions and dead-locks). Additionally, some techniques have been developed specifically for OpenMP to compute and verify data scoping, task dependencies and locks among others [86, 129, 131, 133].

6.2.2 OpenMP hazards for real-time embedded systems

This section analyzes the OpenMP specification to bring forth the features that may jeopardize functional safety. Related work addressing the detection of correctness errors is also included.

6.2.2.1 Unspecified behavior

OpenMP defines the situations that result in an unspecified behavior as: non-conforming programs, implementation-defined features and issues documented to have an unspecified behavior. The impact of each situation to the safety-critical domain, as well as the solutions we propose, are exposed below.

6.2.2.1.1 Non-conforming programs

The OpenMP specification defines several requirements to applications that are parallelized with OpenMP. Programs that do not follow these rules are called *non-conforming*. According to the specification, OpenMP compliant implementations are not required to verify conformity. However, safety-critical environments compel frameworks to do this validation to certify functional safety.

OpenMP restrictions affect directives, clauses and the associated user code. Checking some restrictions just requires the verification of OpenMP constructions (e.g., which clauses and how many times a clause can be associated with a specific directive may be restricted, e.g., *at most one* `if` *clause can appear on the* `task` *directive*). Conversely, checking other restrictions requires

visibility of different parts of the application (e.g., some regions cannot be nested and/or closely nested in other regions, e.g., `atomic` *regions must not contain OpenMP constructs*).

Compilers must implement inter-procedural analysis to have access to the whole application. This capability has been successfully implemented in many compilers following different approaches, such as Intel IPO [70] or GCC LTO [59]. Nevertheless, access to the whole code is possible only for monolithic applications. This is not very common in the critical domain, where systems consist of multiple components developed by different teams, and rely on third-party libraries. In these cases, additional information may be needed. We discuss this situation and propose a solution to it in Section 6.2.3.1. This solution is based on new directives that provide the required information. Henceforward, we assume that the information needed to perform whole program analysis is always accessible.

6.2.2.1.2 Implementation-defined behavior

Some aspects of the implementation of an OpenMP-compliant system are not fixed in the specification. These aspects are said to have an *implementation-defined* behavior, and they may indeed vary between different compliant implementations. The different aspects can be grouped as follows:

1. Aspects that are naturally implementation-defined, so the specification can be used in multiple architectures: definitions for *processor*, *device*, *device address* and *memory model*.
2. Aspects that are implementation-defined to allow flexibility: internal control variables (e.g., *nthreads-var* and *def-sched-var* among others); selection, amount and distribution of threads (e.g., `single` construct); dynamic adjustment of threads; etc.
3. Aspects caused by bad information specified by the user: values out of range passed to runtime routines or environment variables (e.g., the argument passed to `omp_set_num_threads` is not a positive integer).

Aspects in groups 1 and 2 may not lead to an execution error or prevent the program from validating. This is not the case for aspects in group 3, where an implementation may decide to finish the execution if a value is not in the range it was expected to be. Besides, cases in group 2 may result in different outcomes depending on the platform used for the execution. For example, when the `runtime` or the `auto` kinds are used in the `schedule` clause, the decision of how the iterations of a loop is scheduled is deferred until run-time.

In the light of all that, some aspects in groups 2 and 3 are not suitable in a safety-critical environment because they are non-deterministic and may cause an undesired result. Situations such as the application aborting due to an unexpected value passed to either an environment variable or a runtime routine can be solved by defining a default value that prevents the application to end (note that this value can be different across implementations without that affecting functional safety). Situations such as an `auto` or `runtime` value in the `schedule` clause can be solved by taking a conservative approach at compile-time (i.e., if a deadlock may occur for any possible scheduling option, then the compiler will act as if that scheduling happens). Situations such as runtimes defining different default values for ICVs like *nthreads-var* do not need to be addressed, because they do not bring on any hazard regarding functional safety.

6.2.2.1.3 Other unspecified behavior

The rest of situations resulting in an undefined behavior are errors and need to be addressed to guarantee functional safety. These situations can be classified in three groups, depending on the moment at which they can be detected:

1. Situations that can be detected at compile-time. In this case we can distinguish those that can be solved by the compiler (e.g., data-race conditions could be solved by automatically protecting accesses with a `critical` construct or synchronizing the accesses –Section 6.2.2.3 shows more details about data race management–), and those that need user intervention (e.g., compilers should abort compilation and report to the user situations such as the use of non-invariant expressions in a linear clause).
2. Situations that can be detected at run-time. In this case, safety relies on programmers because the results deriving from these situations cannot be handled automatically. Thus, users are compelled to handle errors such as reduction clauses that contain accesses out of the range of an array section, or using the `omp_target_associate_ptr` routine to associate pointers that share underlying storage (Section 6.2.2.5.1 explores error handling techniques).
3. Situations that cannot be detected. These involve the semantics of the program (e.g., a program that relies on the task execution order being determined by a priority-value), and are further discussed in Section 6.2.2.5.2 .

6.2.2.2 Deadlocks

OpenMP offers two ways to synchronize threads: via directives (`master` and synchronization constructs such as `critical` and `barrier`), and via runtime routines (lock routines such as `omp_set_lock` and `omp_unset_lock`). Although both mechanisms may introduce deadlocks, the latter is much more error-prone because these routines work in pairs. Furthermore, OpenMP introduces the concept of nestable locks, which differ from the regular locks in that they can be locked repeatedly by the same task without blocking.

Synchronization directives may cause deadlocks if various `critical` constructs with the same name are nested. Synchronization directives can introduce other problems as well, like enclosing a `barrier` construct in a condition that is special to a thread. Since barriers must always be encountered by all threads of a team, the previous situation will be non-conforming. Conservative compiler analysis (meaning that false positives may appear) can easily catch these errors if whole program analysis is supported.

Locking routines may cause errors in the following situations: attempt to access an *uninitialized* lock, attempt to unset a lock owned by another thread, and attempt to set a simple lock that is in the *locked* state and is owned by the same task. There exist numerous techniques for deadlock detection, such as Chord [103] and Sherlock [47], that apply to different programming models. Most of the approaches pursue scalability without losing accuracy, thus effectiveness. However, safety-critical environments require soundness. In this regard, the only sound approach, to the best of our knowledge, for detecting deadlocks in C/Pthreads programs is the one developed by Kroening et al. [80]. OpenMP simple locks are comparable to Pthreads mutex, so the previous

technique can be extended to OpenMP. Nestable locks have other peculiarities and it may not be possible to detect deadlocks at compile-time. In such a case, they should not be permitted.

The use of untied tasks may cause deadlocks that are nonexistent when using tied tasks. This is because the OpenMP Task Scheduling Constraint (TSC) number 2[1] prevents from certain situations involving tied tasks to cause a deadlock by restricting the tasks that can be scheduled at a certain point. Based on that, using tied tasks may seem more suitable for critical real-time embedded systems. It has been, however, demonstrated that timing analysis for untied tasks is much more accurate than for tied tasks [139]. There is thus a trade-off between functional safety and predictability. For the sake of correctness, untied tasks may be disabled at compile-time only when the static analysis detects that a deadlock caused by untied tasks may occur.

6.2.2.3 Data race conditions

Race conditions appear in a concurrent execution when two or more threads simultaneously access the same resource and at least one of them is a write. This situation is not acceptable for a safety-critical environment since the results of the algorithm are non-deterministic. The problem of detecting data races in a program is NP-hard [105]. On account of this, a large variety of static, dynamic and hybrid data race detection techniques have been developed over the years.

On one hand, dynamic tools extract information from the memory accesses of specific executions. Despite this, there exists an algorithm capable of finding at least one race when races are present, as well as not reporting false positives [14]. On the other hand, static tools still seek a technique with no false negatives and minimal false positives. Current static tools have been proved to work properly on specific subsets of OpenMP such as having a fixed number of threads [93], or using only affine constructs [27]. A more general approach exists to determine the regions of code that are definitely non-concurrent [87]. Although inaccurate, it does not produce false negatives, which is paramount in the safety-critical domain. Therefore, the previously mentioned techniques can be combined to deliver conservative and fairly accurate results.

6.2.2.4 Cancellation

OpenMP 4.0 incorporates the cancellation constructs (i.e., `cancel` and `cancellation point`), which allow jumping to the end of a parallel computation at a certain point within that region. Unlike other models such as the Pthreads, OpenMP only accepts synchronous cancellations at cancellation points. Although this eliminates resource leak risks, the technique introduces non-determinism, which is not desirable in safety-critical environments. Due to the use of cancellation constructs, non-determinism appears in the following situations:

1. The order of execution between one thread that activates cancellation and another thread that encounters a cancellation point.
2. The final value of a reduction or lastprivate variable in a canceled construct.
3. The behavior of nested regions suitable of being canceled.

[1] OpenMP TSC 2 states that *“scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a barrier region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendent task of every task in the set”*.

If a code is well written, case 1 may only affect performance, but the code will still deliver a valid result whether cancellation occurs or not. Case 2, instead, may lead to errors if some threads have not finished their computation. Nonetheless, static analysis can verify that reduction and lastprivate variables are not used within a construct that may be subject to cancellation, or that the variables are used only when no cancellation occurs. Finally, case 3 can be solved by statically verifying that regions subject to cancellation are not nested.

Another issue arises when locks are used in regions subject to cancellation, because users are responsible for releasing those locks. Current deadlock detection techniques do not take into account the semantics of the cancellation constructs. Nonetheless, these techniques can easily be adopted because the effect of a cancellation is similar to the existence of a jump to the end of the region.

6.2.2.5 Other features to consider

Although they do not necessarily entail a hazard, there are other issues that are worth mentioning in the context of this study. These are explored in this section, and include error handling techniques, semantic mistakes and nested parallelism.

6.2.2.5.1 Error handling

Resiliency is a crucial feature in safety-critical domains. However, OpenMP does not prescribe how implementations must react to situations such as the runtime not being able to supply the number of threads requested, or the user passing an unexpected value to a routine. While the former is a problem caused by the runtime environment, the latter is an error produced by the user. Both eventually become an unspecified behavior according to the specification, but they can be addressed differently. On one hand, if the error is produced by the environment, users may want to define what recovery method needs to be executed. On the other hand, errors produced by the user are better caught at compile-time or handled by the runtime (e.g., static analysis techniques for data-race and deadlock detection).

Several approaches have been proposed with the aim of adding resiliency mechanisms to OpenMP. There are four different strategies for error handling [164]: exceptions, error codes, call-backs and directives. Each technique can be applied according to its features to different languages and situations. Exception based mechanisms fit well in programs exploiting the characteristics of exception-aware languages (e.g., C++, Ada) [49]. Error code based techniques are a good candidate when using a language unaware of exceptions (e.g., C, Fortran). Call-back methods have the advantage of isolating the code that is to be executed when an exception occurs, and thus enhance readability and maintainability [46]. Finally, the use of specific OpenMP directives has the advantage of being simple, although they cannot cover all situations and users cannot define an exact behavior. The latter is the only approach already adopted in the specification with the cancellation constructs (see more details in Section 6.2.2.4).

A safety-critical framework supporting OpenMP will require the implementation of error-handling methodologies in order to ensure functional safety.

6.2.2.5.2 Semantics of OpenMP

For an analysis tool, it is possible to address correctness based on how the program is written. However, addressing whether the program behaves as the user wants is another matter altogether. This said, some features of OpenMP may be considered as hazardous because their use may result in errors involving the semantics of the program. We discuss some of them as follows:

- A program that relies on a specific order of execution of the tasks based on their priorities is non-conforming.
- When and how some expressions are to be executed is not defined in OpenMP. For example: whether, in what order, or how many times any side effects of the evaluation of the `num_threads` or `if` clause expressions of a `parallel` construct occur is unspecified; likewise, the order in which the values of a reduction are combined is unspecified as well. Thus, an application that relies on any ordering of the evaluation of the expressions mentioned before is non-conforming.
- The storage location specified in task dependences must be identical or disjoint. Thus, runtimes are not forced to check whether two task instances have partially overlapping storage (which eases considerably the implementation of the feature in the runtime).
- The use of flushes is highly error-prone, and makes it extremely hard to test whether the code is correct. However, the use of the `flush` operation is necessary for some cases such as the implementation of the producer-consumer pattern.

Frameworks cannot prevent users from writing senseless code. However, some of the features mentioned before could be deactivated if the level of criticality demands it. It is a matter of balance between functionality and safety. Thus, if necessary, support for task priorities and the `flush` directive could be deactivated. The case regarding side-effects could be simplified to using associative and commutative operations in reductions, and expressions without side-effects in the rest of clauses. Finally, the case regarding dependence clauses could be solved at run-time by resuming parallel execution (i.e., initiate sequential execution) when a task contains non-conforming expressions in its dependence clauses, although this solution causes a serious impact in the performance of the application.

6.2.2.5.3 Nested parallelism

OpenMP allows nesting parallel regions to get better performance in cases where parallelism is not exploited at the same level. A distributed shared-memory machine with an appropriate memory hierarchy is necessary to exploit the benefits of this feature (the major HPC architectures).

The nature of critical real-time embedded systems is quite different, where both memory size and processor speed are usually constrained. Furthermore, the use of nested parallelism can be costly due to the overhead of creating multiple parallel regions, possible issues with data locality, and the risk of oversubscribing system resources. For the sake of simplicity, and considering that current embedded architectures will not leverage the use of nested parallelism, this feature could be deactivated by default.

6.2.3 Adapting the OpenMP specification to the real-time domain

This section presents our proposal to enable the use of OpenMP in safety-critical environments without compromising functional safety. It is based on the discussion in Section 6.2, and the proposal can be divided in two facets: different changes to the specification, and a series of compiler and runtime implementation considerations.

6.2.3.1 Changes to the specification

As we introduce in Section 6.2.2.1, whole program analysis may not be enough if the system includes multiple components developed by different teams, or make use of third-party libraries implemented with OpenMP. In such a case, we propose that these components or libraries augment their API with information about the OpenMP features used in each method. As a result, compilers will be able to detect situations such as illegal nesting of directives and data accessing clauses (i.e., data-sharing attributes, data mapping, data copying and reductions), data-race conditions and deadlocks even when the code of all components is not accessible at compile time.

To tackle illegal nesting and deadlocks, we propose to add a new directive called `usage`. This directive is added to a function declaration and followed by a series of clauses. The clauses determine the features of OpenMP that are used within the function and any function in its call graph, and can cause an illegal nesting. Overall, the clauses that can follow the pragma `usage` are one of the following:

- Directive related: `parallel`, `worksharing` (which epitomizes `single`, `for`/`do`, `sections` and `workshare`), `master`, `barrier`, `critical`, `ordered`, `cancel`, `distribute_construct` (which epitomizes `distribute`, `distribute simd`, distribute parallel loop and distribute parallel loop SIMD), `target_construct` (which epitomizes `target`, `target update`, `target data`, `target enter data` and `target exit data`), `teams`, `any` (which epitomizes any directive not included in the previous items).
- Clause related: `firstprivate`, `lastprivate`, `reduction`, `map`, `copyin` and `copyprivate`.

Based on the restrictions that apply to the nesting of regions (Section 2.17 of the specification [113]) and the restrictions that apply to the mentioned data accessing clauses, Algorithm 7 extracts the set of rules that define when the a specific directive or clause has to be added to the list of clauses of the directive `usage`.

To avoid data races, we propose to add a new directive called `globals`. This directive, added to a function declaration, defines which data is used within the function while it can be accessed concurrently from outside the function, thus producing a data-race. Different clauses accompany this directive: `read`, `write`, `protected_read` and `protected_write`, all accepting a list of items. While `read` and `protected_read` must be used when global data is only read, `write` and `protected_write` are required when global data is written, independently of it being read as well. The *protected* versions of these clauses must be used when the access is within an `atomic` or a `critical` construct.

Algorithm 7 Rules to determine the clauses of the `usage` directive to be added to the contract of a safety-critical function.

- Clauses `parallel`, `worksharing`, `master`, `barrier` and `ordered` are required when the corresponding construct is the outermost construct.
- Clauses `critical` and `target_construct` are required if there is any occurrence of the corresponding construct.
- Clause `teams` is required if the corresponding construct is orphaned.
- Clauses `cancel` and `cancellation point` are required if the corresponding constructs are not nested in their corresponding binding regions.
- Clause `any` must be specified if OpenMP is used and no previous case applies.
- Data accessing clauses are required when they apply to data that is accessible outside the application, and particular constraints apply to them:
  * Clause `firstprivate` is required if used in a worksharing, `distribute`, `task` or `taskloop` construct not enclosed in a `parallel` or `teams` construct.
  * Clauses `lastprivate` and `reduction` are required if used in a worksharing not enclosed in a `parallel` construct.
  * Clauses `copyin`, `copyprivate` and `map` are required in any case.

Listings 6.1 and 6.2 illustrate the use of the two mentioned directives. The former contains the definition of function *foo*, which uses an essential feature for the use of OpenMP in parallel heterogeneous embedded architectures: the `target` construct. The function defines an asynchronous target task that offloads some parallel computation (spawned in the `parallel` construct and distributed in the `for` construct) to a device. The parallel computation within the device is synchronized using the `critical` construct, and is canceled if the `cancel` directive is reached. The latter contains the declaration of function *foo*, augmented with the `usage` and `globals` directives. All possible clauses associated with these directives are explained as follows:

- Clauses `target_construct` and `critical` associated with directive `usage` indicate that the function executes one or more `target` and `critical` constructs. A programmer and/or compiler can avoid calling function *foo* from within a `target` or a `critical` construct, thus avoiding an illegal nesting or even deadlocks.
- Clause `map` associated with directive `usage` indicates the variables that are mapped to/from a target device. A programmer and/or compiler can avoid mapping threadprivate variables, which is forbidden in the specification.
- The `usage` directive does not contain any other clause for the following reasons: clause `cancel` is not included because it is nested in its binding region, clauses `task` and `parallel for` are not included because no rule apply to them, and clause `firstprivate` is not included because it does not concern to data that is visible from outside the function.
- Clauses `write` and `protected_write` associated with directive `globals` indicate that variables *arr[0:N-1]* and *sum* are both written, being *sum* written within a synchronization construct. This information allows determining if the variables are in a race condition without analyzing the function, and therefore synchronize the accesses to the variables appropriately.

```
1  void foo(float* arr, unsigned N, unsigned M,
2           float &sum, float MAX_SUM) {
3     #pragma omp target map(tofrom:arr[0:N-1], sum) \
4                        firstprivate(N, M, MAX_SUM) nowait
5     #pragma omp parallel
6     #pragma omp for
7     for (int i=0; i<N; ++i) {
8        arr[i] = bar(i);
9        if (i % M == 0) {
10          #pragma omp critical
11          sum +=arr[i];
12       }
13       if (sum > MAX_SUM) {
14          #pragma omp cancel for
15       }
16    }
17 }
```

Listing 6.1: Example of OpenMP function using several constructs.

```
1  #pragma omp usage target_construct critical map(tofrom:arr[0:N-1])
2  #pragma omp globals write(arr[0:N-1]) protected_write(sum)
3  void foo(float* arr, unsigned N, unsigned M,
4           float &sum, float MAX_SUM);
```

Listing 6.2: Function declaration of method in Listing 6.1 using the proposed extensions for safety-critical systems.

Listings 6.3 and 6.4 show another example of the proposed directives. In this case, the function definition in the former listing performs the *factorial* computation parallelized using the `for` worksharing; and the function declaration in the latter listing shows the clauses required for the method to be used in a functional safe environment. Clause `any` is specified because no rule applies to directive `for`, and clause `reduction` is specified because the reduction is used in a worksharing not enclosed in a parallel region. With this information a programmer and/or compiler can check whether the variable being reduced is shared in the parallel regions to which any of the worksharing regions bind. Analysis may also verify if the *factorial* function is not called from within an atomic region, thus causing the program to be non-conforming. Finally, race analysis can detect whether the variable *factorial* is in a race condition by means of the clause `write`.

```
1  void factorial(int N, int &fact) {
2    fact = 1;
3    #pragma omp for reduction(*:fact)
4    for(int i=2; i <= N; ++i)
5      fact *= i;
6  }
```

Listing 6.3: Factorial computation parallelized with OpenMP.

```
1  #pragma omp usage any \
2                    reduction(factorial)
3  #pragma omp globals write(factorial)
4  void factorial(int N, int &factorial);
```

Listing 6.4: Function declaration for method in Listing 6.3 using the extensions for safety-critical OpenMP.

6.2.3.2 Automatic definition of the contracts of a safety-critical OpenMP library

The use of the `usage` and the `globals` directives is a promise that something happens (either the existence of a particular construct, or the use of a global variable), not only in the function where the directive is placed, but in any possible path reachable from within that function. These contracts are to be defined by the programmer when the code of the application will no be reachable to others. However, for the application to be safe, compiler analysis techniques are required to certify its correctness considering the proposed directives. For this reason, inter-procedural whole-program analysis together with static call graph generation[2] and use-definition analysis are necessary in order to check whether the contracts are correct, or to automatically determine these contracts.

The process to define the contracts (i.e., the `usage` and `globals` directives) of the API of a safety-critical library is depicted in Algorithm 8. The methodology consists on generating the call graph of each method that is visible from outside the library. Then, each call graph is traversed from the leaves to the roots (avoiding cycles) and, for each node: 1) propagates to the current node the directives computed in the called nodes, and 2) computes the information of the current node to purge the propagated information and define the final information of the current node.

Algorithm 8 High-level algorithm to compute the contracts of a safety-critical OpenMP library.

1: **for each** call graph whose root can be called from outside the library **do**
2:     **for each** node in the call graph (traversed from leaves to roots) **do**
3:         Propagate the `usage` and `globals` directives from the called nodes.
4:         **for each** OpenMP directive used within the function **do**
5:             **if** it allows removing some clause propagated from called nodes to the `usage` directive **then**
6:                 Remove the corresponding clauses from the `usage` directive.
7:             **end if**
8:         **end for**
9:         **for each** OpenMP directive and clause used within the function **do**
10:             Follow the rules defined in Algorithm 7 to decide if the directive/clause has to be added to the `usage` directive.
11:         **end for**
12:         Add all variables that can be used outside the function (parameters by reference, parameters with pointer type, and variables shared across applications -e.g., *errno*-) to the `usage` directive.
13:     **end for**
14: **end for**

As an illustration, Figure 6.1 shows the propagation of the `usage` and `globals` directives over the call graph of a snippet of an application. Consider method *library_entry* as the entry point of the application, hence the method to augment with the contract. Function *A* defines the usage of the `master` directive (because it is the outermost construct) and the `critical` directive (because it just appears). Additionally, this method also specifies that the variable *s* is read and

[2] A *call graph* is a type of control flow graph which represents calling relationships between subroutines in a program.

written within a synchronization construct. Function *B* defines the usage of the `cancel` construct because it does not appear within its binding region. When this information is propagated to the function *library_entry*, the `master` and the `cancel` constructs disappear from the list of clauses of the `usage` directive because the reason for them to be there does not fulfill anymore. The `globals` information is propagated because variable *s* is visible outside the function. Finally, the information of the node itself is added to the final contract of the function including the `parallel` clause to the `usage` directive.

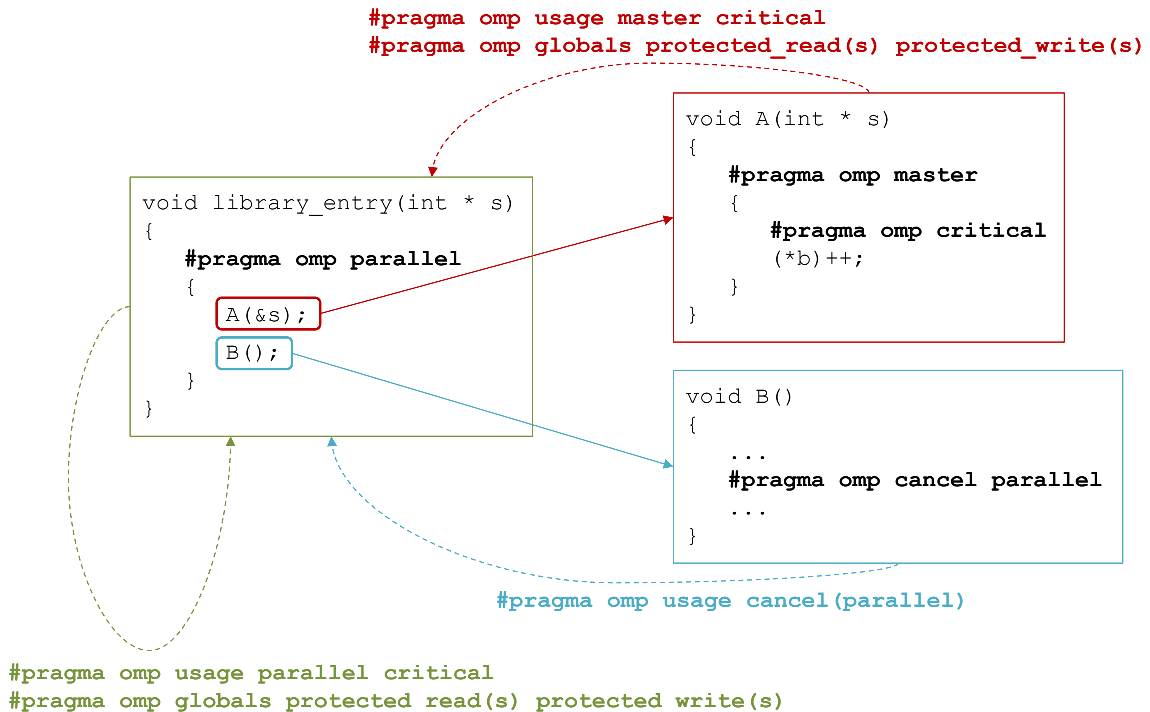


Figure 6.1: Example of propagation of the usage and global directives over a call graph.

6.2.3.3 Implementation considerations

Both compilers and runtimes used in critical systems must be qualified following the relative functional safety standard, e.g., ISO26262 for automotive or DO178C for avionics, to preserve functional safety. The following paragraphs introduce which constraints apply in our case.

6.2.3.3.1 Compiler contract

The development tools used for critical real-time systems need to qualify to the same integrity level[3] as the application they are helping to develop. Features such as determinism, correctness, robustness, and conformance to standards are considered for qualification. Nonetheless, current guidelines make the qualification of development tools very difficult [78]. As an example, the standard for Software Considerations in Airborne Systems and Equipment Certification

[3]The *integrity level*, also called *criticality level*, refers to the consequences of the incorrect behavior of a system. These levels are defined in different scales such as the *Safety Integrity Level* (SIL) for automotive and the *Development Assurance Level* (DAL) for avionics.

(DO-178C) [44] reads: "Upon successful completion of verification of the software product, the compiler is considered acceptable for that product". As a result, sometimes compilers do not need to be qualified. Nonetheless, to gain assurance, some characteristics must be incorporated, such as being fully tested for complete coverage analysis[4], and being used in the same configuration, options and environment as the one used to compile any other objects related to the application.

However, for an OpenMP compiler to be valid in a critical real-time environment, it must ensure the source code is compliant with the OpenMP specification. For that reason, the compiler must implement the necessary analysis techniques to allow whole program analysis. Additionally, the compiler must also include specific and sound techniques for data-race and deadlock detection, as well as the correctness analysis that allows statically detecting and fixing the unspecified behaviors commented in Section 6.2.2.1.

6.2.3.3.2 Runtime contract

As a result of the analysis presented in Section 6.2, we conclude that runtime libraries used in safety-critical environments shall follow some requirements to avoid unexpected aborts and fix some programmer errors. The following list is a starting point for these systems to address such undesired results:

- Runtimes should define a default value for all environment variables. This value shall be used when the value specified in the application is out of range, e.g., `OMP_NUM_THREADS` could be 1 by default, and `OMP_NESTED` could be false.
- Some clauses, such as `num_threads` and `device`, take a number as a parameter that must evaluate to a positive integer. Runtimes should define the value to be used if the expression is out of range, for example, 1.
- Other errors can be caught and fixed at run-time, e.g., different instances of the same task or sibling tasks expressing dependence clauses on list items which storage location is neither identical nor disjoint may be executed sequentially.

6.2.4 Conclusion

OpenMP is increasingly being considered as a suitable candidate to be used in critical real-time embedded systems considering its benefits: programmability, portability and efficiency, among others. However, such systems impose strict constraints to ensure safety in terms of functional correctness and time predictability.

This section is focused on functional safety, and proves that most features of OpenMP can be used without compromising functional safety, as long as compilers implement a comprehensive analysis that can prevent errors such as dead-locks and race conditions. Indeed, analysis must involve the entire program, and this can be a challenging scenario. To ease this, we propose to add some new directives that allow whole program analysis even when third-party libraries are used. The majority of the unspecified behaviors defined in the specification can be solved at compile time either automatically by the compiler (e.g., synchronizing variables that otherwise

[4] *Code coverage* is a measure used to describe the amount of the source code of a program being executed when a particular test suite runs.

could be accessed after their life-time has ended), or by the programmer (e.g., the use of non-invariant expressions in a linear clause). Other issues can be successfully addressed at runtime (e.g., unexpected values passed to environment variables and runtime libraries can be solved by defining default values to be used in such cases). In some cases, supporting the required level of criticality might incur more overhead than a traditional OpenMP implementation (e.g., tracking the overlapping among task dependencies). Last but not least, there are a series of features that can be used erroneously if their semantics are not properly exploited (e.g., tasks priorities or flushes). We conclude that support for these features can be deactivated if the level of criticality requires so.

The small modifications we propose back up OpenMP's safety. Nonetheless, there are some lacks in the current specification, e.g., error handling techniques to improve resiliency. Furthermore, despite we deeply address the functional safety aspect, the same analysis concerning time predictability, including starvation, remains as future work.

6.3 Application of OpenMP to a safe language: Ada

This section evaluates the use of OpenMP with Ada at two different levels: 1) using OpenMP as a runtime to run the Ada parallel model, and 2) using raw OpenMP in Ada codes to further exploit unstructured parallelism and heterogeneous architectures. With such a purpose, we first introduce the recently proposed Ada parallel model. Then, we analyze the compatibility between this model and OpenMP. Finally, we analyze the performance of OpenMP with Ada.

6.3.1 Related work

As we introduce in Section 2.1.3, Ada supports a concurrency model based on Ada tasks (independent threads of control) and a set of language mechanisms for inter-task communication (i.e., protected objects). The rationale is that providing language concurrency mechanisms, the compiler has valuable information on the tasking behavior, and this allows building safer programs. The Ada concurrency model is mainly suitable for coarse grain parallelism. For that reason, there has been a significant effort to add support for fine grain parallelism to Ada.

On one hand, there is a proposal to extend the Ada core with extensions that support structured parallelism in the form of parallel blocks and parallel loops (including reductions) [120]. This technique is based on the notion of *tasklets* [98], which are concurrent logical units within an Ada task. Adding parallelism also means adding a source of errors (due to concurrent accesses to global data and synchronizations). For this reason, the mentioned proposal addresses safety using new annotations that enable the compiler to detect data race conditions and blocking operations[5].

On the other hand, there is a user-level library, Paraffin [101, 116], that consists of a set of generic Ada libraries that dynamically manage fine grain parallelism, incorporating mechanisms for parallel loops and reductions, parallel blocks and recursive parallelism. This library provides parallelism managers following work-sharing, work-stealing and work-seeking approaches, on top

[5] *Blocking operations* are defined in Ada to be one of the following: entry calls; select, accept, delay and abort statements; task creation or activation; external calls on a protected subprogram with the same target object as that of the protected action; and calls to a subprogram containing blocking operations.

of pools of worker tasks. Using Ada generics, it provides a simple interface to create and manage parallel execution, and delivers comparable performance to OpenMP or Cilk [29] on structured parallelism for a small number of cores [100].

The proposed extensions, which are currently under discussion and may evolve before being included in the standard (maybe in Ada202X) target only structured parallelism, based on a fully strict fork-join model, on shared memory architectures. For that reason, introducing more advanced parallel programming models to Ada can benefit the exploitation of more complex (unstructured) parallelism and the use of heterogeneous computation. In that respect, Section 6.2 already demonstrate that OpenMP provides the safety properties required by Ada.

6.3.2 Analysis of the Ada and OpenMP parallel models

6.3.2.1 Forms of parallelism

The Ada tasklet model and OpenMP implement a *fork-join* execution model where parallelism is spawned when a *parallel statement* (in Ada) or a *parallel construct* (in OpenMP) is reached, and it is joined at the end of the parallel region. Both models define *execution containers*, named *executor* in Ada and *thread* in OpenMP, and managed by the respective runtimes.

The Ada parallel model introduces two new statements to the language in charge of spawning and distributing the parallel work to executors. These statements allow defining three forms of parallelism: parallel blocks, parallel loops, and reductions. All three mechanisms define a form of structured parallelism, and are defined as follows:

- The *parallel block* statement allows defining several blocks of code that can execute in parallel. Listing 6.5 shows the syntax of this statement.
- The *parallel loop* statement denotes that loop iterations can execute in parallel. In a parallel loop, both the compiler and the runtime are given the freedom to chunk iterations. Although not mandatory, programmers may gain control by defining chunk sizes. Listing 6.6 shows the syntax of this statement. Additionally, the concept of parallel array is introduced to define data being updated within a parallel loop. The syntax is shown in Listing 6.7, where the use of <> indicates an array of unspecified bounds. In that case, the compiler may choose the size based on the number of chunks chosen for the parallelized loops where the array is used. Alternatively, the programmer may provide a bound, thus forcing a specific partitioning.

```
1 parallel
2     x := a * a;
3 and
4     y := b * b;
5 end parallel;
6 res := x + y;
```

Listing 6.5: Ada syntax for parallel blocks.

```
1 for I in parallel lb..ub loop
2     a(I) := a(I) + b(I);
3 end loop;
```

Listing 6.6: Ada syntax for a parallel loop.

```
1 Arr : array (parallel <>) of a_type
2     := (others => initial_value);
```

Listing 6.7: Ada syntax for a not chuncked parallel.

- *Reductions* are defined to be an operation for values in a parallel array that consists in combining the different values of the array at the end of the processing with the appropriate reduction operation. The syntax for parallel reductions is still under discussion [119] and the current proposal is to define the reduction in the type, as in Listing 6.8.

```
1  ...
2     type Partial_Array_Type is new array (parallel <>) of Float;
3     with Reducer => "+", Identity => 0.0;
4     Partial_Sum : Partial_Array_Type := (others => 0.0);
5     Sum : Float := 0.0;
6  begin
7     for I in parallel Arr'Range loop
8         Partial_Sum(<>) := Partial_Sum(<>) + Arr(I);
9     end loop;
10    Sum := Partial_Sum(<>)'Reduced; -- reduce value either here or
11                                    -- during the parallel loop
12 ...
```

Listing 6.8: Parallel reduction with proposed Ada extensions.

A transfer of control[6] or exception[7] within one parallel sequence (in a parallel loop, each chunk is treated as a separate sequence) aborts the execution of parallel sequences that have not started, and potentially initiates the abortion of those sequences not yet completed[8]. Once all parallel sequences complete, then the transfer of control or exception occurs.

Unlike Ada, OpenMP splits the spawning and distribution of parallel work in different statements: the `parallel` construct spawns work, and several constructs distribute this work to threads. The constructs for distribution can be classified in two different models:

- The *thread-centric* model exploits structured parallelism distributing work by means of work-sharing constructs. It provides a fine grain control of the mapping between work and threads. The most representative constructs are `for` and `sections`.
- The *task-centric* model exploits both structured and unstructured parallelism distributing work by means of tasking constructs. It provides a higher abstraction level in which threads are fully controlled by the runtime. The most representative constructs are `task` and `taskloop`.

The two models have comparable performance [122]. Listings 6.9 and 6.10 are the equivalent to Listings 6.5 and 6.6 using the OpenMP tasking model. The notation is adapted to fit the syntax of Ada: a) since Ada already defines pragmas of the form `pragma Name (Parameter List);`, we propose to introduce a new kind of pragma `OMP` together with the directive name (e.g., `task`, `barrier`, etc.), and b) we follow the syntax of Ada to group sequences of statements (i.e., use a closing statement to match the beginning of the group instead of brackets).

Figure 6.2 illustrates the flexibility of the OpenMP fork-join model compared to that of Ada. Due to the separation of the spawn and distribution operations, OpenMP allows executing simultaneously several constructs: the example in the figure shows two parallel loops executing

[6] A *transfer of control* causes the execution of a program to continue from a different address instead of the next instruction (e.g., a return instruction).

[7] *Exceptions* are anomalous conditions requiring special processing. Ada has predefined exceptions (language-defined run-time errors) and user-defined exceptions.

[8] The rules for abortion of parallel computations are still under discussion [119].

```
1 pragma OMP (parallel);
2 pragma OMP (single);
3 begin
4   pragma OMP (task);
5      x := a * a;
6   pragma OMP (task);
7      y := b * b;
8 end;
9 res := x + y;
```

Listing 6.9: OpenMP syntax for parallel blocks.

```
1 pragma OMP (taskloop);
2 for I in range lb..ub loop
3    a[I] := a[I] + b[I];
4 end loop;
```

Listing 6.10: OpenMP syntax for a parallel loop.

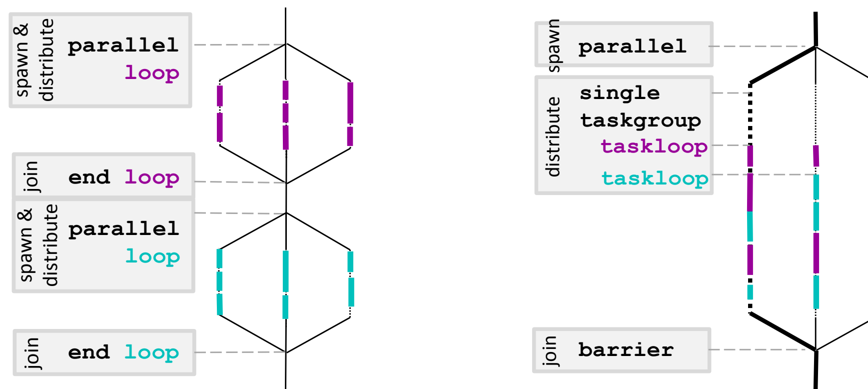


Figure 6.2: Concurrency available with the Ada parallel model (left) and OpenMP tasks (right).

concurrently due to the use of the `taskgroup` directive, which avoids the implicit barrier after the first taskloop. This feature can potentially increase parallelism and reduce unnecessary synchronizations. Besides, in OpenMP the thread that spawns work may not be the same as the one that distributes it, while in Ada, the sames thread does the two operations.

6.3.2.2 Execution model

The Ada parallel model defines the *tasklet* as the unit of parallelism. Tasklets come into existence when the parallel work starts, and terminate at the end of the parallel work. With this, the Ada execution model is based on a limited form of *run-to-completion* where tasklets are typically executed by a unique executor, unless they perform an operation that requires blocking or suspension; at these points, the tasklet is allowed to migrate to a different executor. Note that, even if the tasklet does not change executor, it is not mandatory for it to run uninterruptedly or to execute in the same core, since executors may be scheduled in a preemptive scheduler.

The concept of tasklet is very similar to the concept of OpenMP task. First, both are containers that enable fine grain parallelism. Second, the existence of the container is limited to the work it encloses. Third, and most important, OpenMP tasks, as Ada tasklets, can be prioritized and preempted. In that regard, OpenMP defines a Task Scheduling Point (TSP) as the moment at which a thread can stop executing a specific task and start executing a different one[9]. The runtime

[9]OpenMP associates TSPs to different points in a program, e.g., after the generation of an explicit task (see the

is responsible of deciding whether the task being executed is preempted (and potentially migrated) or not. Similarly to Ada, OpenMP tasks (both tied and untied) are not forced to run uninterruptedly.

Furthermore, there are two final considerations to take into account: 1) the main difference between Ada tasklets and OpenMP tasks is that OpenMP allows users to explicitly define tasks whereas in Ada, tasklets are transparent, and 2) the equivalence with the OpenMP thread-centric model is not straight-forward because OpenMP maps the logical concurrent units of work to threads directly, and neither the specification nor the runtime provide any feature for preempting work-sharings.

6.3.2.3 Use of resources

OpenMP allows programmers to define the amount of computing resources to be used in a parallel region by means of the `num_threads` clause attached to the `parallel` construct. If none is defined, then the number is implementation-defined (although the number of cores is commonly considered).

For the Ada parallel model, it is still not defined if the programmer can control the number of executors assigned to a parallel region, although a mechanism shall exist to control the number of executors per Ada task. In this direction, the current proposal defines three kinds of *parallel progression model*. Ada denotes that the parallel execution *progresses* if at least one of the spawned tasklets is being executed by an executor:

- *Immediate progress*. Ready tasklets can always execute if there are available cores.
- *Eventual progress*. Ready tasklets may have to wait for the availability of an executor even if cores are available, but it is guaranteed that one executor will become available so that the tasklet will eventually be executed.
- *Limited progress*. Ready tasklets may have to wait for the availability of an executor even if cores are available, and it is not guaranteed that one executor will eventually become available. This may happen when there is a limited number of executors and all are blocked.

Note that runtimes only need to support one such model. The two first cases guarantee progression for any program, even if the runtime does not support tasklet migration between executors when tasklets block. The third one requires static analysis to determine the tasks neither starve nor deadlock, and it is suitable when the resources of the program and the runtime structures are statically determined.

The OpenMP specification does not impose any model of progression, as it is responsibility of the programmer to guarantee that the execution neither stalls nor starves. However, the execution model enables to mimic progression defined by Ada, as will be explained in Section 6.3.3.2.

6.3.2.4 Memory model

The memory model defined for tasklets is based on that of the Ada base language. Ada does not define a *memory model* as such, instead it defines specific types and subclauses that allow

complete list in Section 2.9.5 of the specification [113]). The language also defines the directive `taskyield` to explicitly introduce a TSP.

describing how shared data is accessed by the different threads. These are: a) *protected* objects (provide a mutual exclusion mechanism to access data items), b) *volatile* objects (force all tasks of the program that read or update the object to see the same order of updates), and c) *atomic* objects (force all reads and updates of the object as a whole to be indivisible). Furthermore, the initial proposal of Ada tasklets considers that all variables within a parallel section should be volatile for the sake of simplicity. However, this may be too costly, and a different approach could be the introduction of data-sharing attributes, either computed by the compiler (improving safeness) or specified by the user (allowing for finer accuracy).

On the other hand, OpenMP defines a relaxed-consistency memory model with three different types of memory: 1) the *memory*, where all threads have access to, 2) the *temporary view* of each thread, that eventually may be consistent with the *memory*, and 3) the *threadprivate view* of each thread, which cannot be accessed by other threads. The *flush* operation allows for consistency among the different views of the memory. The operation can be explicitly requested by the user, by means of the `flush` directive, or implicitly forced by the programming model (i.e., OpenMP introduces implicit flushes at strategic points in the code such as barrier regions, and the entry and the exit of `atomic` operations.

In OpenMP, the visibility of the variables can be defined using three different approaches: 1) apply the default data-sharing attributes defined in the specification and based in the storage of the variables; 2) manually define the visibility be means of data-scoping clauses (i.e., `shared`, `firstprivate`, `lastprivate` and `private`); and 3) use the auto-scoping technique [129] to automatically determine the visibility based on the usage and liveness of the variables. The auto-scoping technique is a sound mechanism to determine the data-sharing attributes of a tasklet, as they serve to determine the attributes of an OpenMP task. Since the memory model of the Ada tasklet model is not final, the current analysis indicates that, so far, OpenMP is a candidate to mimic the Ada tasklet model. Also considering that protected objects can always be used as shared within an OpenMP task, hence releasing OpenMP from the duty of managing the consistency of those variables.

6.3.2.5 Safety

Despite the clear benefits of parallel computation in terms of performance, parallel programming is complex and error prone, and that may compromise correctness and so safety. Hence, it is of paramount importance to incorporate compiler and run-time techniques that detect errors in parallel programming.

There are two main sources of errors when dealing with parallel code: a) the concurrent access to shared resources in a situation of race condition, and b) an error in the synchronization between parallel operations leading to a deadlock. To guarantee safety, Ada parallel code must use atomic variables and protected objects to access shared data. Moreover, the compiler shall be able to complain if different parallel regions might have conflicting side-effects.

In that respect, due to the difficulty of accessing the complete source code to perform a full analysis, the proposed Ada extensions suggests a two-fold solution [147]: a) address

race conditions by adding an extended version of the SPARK `Global` aspect to identify the memory locations that are read and written, and b) address deadlocks by the defined execution model, together with a new aspect called `Potentially_Blocking` that indicates whether a subprogram contains statements that are potentially blocking.

In the same line, as we propose in Section 6.2.3.1, directives `globals` and `usage critical` may allow identifying potential data races and deadlocks when third-party code is used. These directives have been proposed to cover the lack of support provided by C/C++ and Fortran. Hence, they are not needed when using OpenMP with Ada, as the previously mentioned aspects can be used.

6.3.3 Supporting the Ada parallel model with OpenMP

This section further analyses the OpenMP and Ada execution models, and demonstrates that OpenMP is a firm candidate to implement Ada parallel blocks and loops statements.

6.3.3.1 Preemption

As we introduce in Section 6.3.2.2, the limited form of run-to-completion implemented in the tasklet model is mappable to the OpenMP tasking model. The points where a tasklet can be preempted (at blocking or suspension) can be implemented using the OpenMP `taskyield` operation.

The OpenMP tasking model defines two different types of tasks: tied and untied (see details in Section 2.1.1.1). Untied tasks are more suitable to implement tasklets, because this model allows tasks to migrate between threads. Moreover, untied tasks have better time predictability than tied tasks, due to their work-conserving nature [139].

6.3.3.2 Progression Model

The OpenMP specification does not impose any model of progression, however it supports progress as defined in the Ada parallel model. Although the OpenMP runtime cannot dynamically modify the number of threads in a team (and therefore it cannot create a new thread when a task blocks), it can move blocked tasks to a waiting queue and reuse threads to execute other tasks. To implement immediate progress, the OpenMP runtime must enforce a work-conserving scheduler, and the number of threads assigned to parallel regions must be bigger or equal than the number of cores. This way, whenever there are resources available, tasks will be scheduled.

Note that OpenMP tied tasks are not suitable to implement immediate progress due to the non-work-conserving nature of the scheduler, but even in this case eventual progress is possible, as long as threads are reused when tasks block. The same happens if the number of threads is smaller than the number of cores.

6.3.3.3 Fork-join Model

The fully strict fork-join model required by the Ada parallel model is fully supported by OpenMP. Since OpenMP does not force the distribution of work to be done at the same point as the spawn of parallelism, explicit synchronizations may be needed. This is the case when implementing nested parallelism in Ada. Figure 6.3a presents a code snippet with nested parallelism using Ada nested parallel blocks, which spawns and distributes twice (at lines 1 and 3). This code can be transformed in two ways using the OpenMP tasking model: 1) using nested parallel regions as shown in Figure 6.3c, which supposes spawning parallelism twice as well (lines 1 and 7), and 2) using nested tasks as shown in Figure 6.3b, which supposes spawning parallelism just once (line 1), and requires a taskwait before *code 4* to force synchronization of the inner block.

```
1  parallel
2    -- code 1
3    parallel
4      -- code 2
5    and
6      -- code 3
7    end parallel
8    -- code 4
9  and
10   -- code 5
11 end parallel;
```

(a) Ada

```
1  pragma OMP (parallel);
2  pragma OMP (single);
3  begin
4    pragma OMP (task, untied);
5    begin
6      -- code 1
7      pragma OMP (task, untied);
8        -- code 2
9      pragma OMP (task, untied);
10       -- code 3
11     pragma OMP (taskwait);
12     -- code 4
13   end;
14   pragma OMP (task, untied);
15     -- code 5
16 end;
```

(b) OpenMP with nested tasks

```
1  pragma OMP (parallel);
2  pragma OMP (single);
3  begin
4    pragma OMP (task, untied);
5    begin
6      -- code 1
7      pragma OMP (parallel);
8      pragma OMP (single);
9      begin
10       pragma OMP (task, untied);
11         -- code 2
12       pragma OMP (task, untied);
13         -- code 3
14     end;
15     -- code 4
16   end;
17   pragma OMP (task, untied);
18     -- code 5
19 end;
```

(c) OpenMP with nested parallels

Figure 6.3: Mapping nested parallelism between Ada and OpenMP.

The Ada tasklet model does not specify how the runtime manages resources of parallel executions, therefore both transformations are possible. The version shown in Figure 6.3b may reduce the overhead of creating and destroying an extra team of threads. However, it is interesting to have the possibility of exploiting two different levels of parallelism for those cases where the parallelism is not exposed at the same level, or where there are load balancing problems.

6.3.4 Supporting the OpenMP Tasking Model in Ada

Section 6.3.3 proposed OpenMP as an implementation of the Ada parallel model. This section evaluates the use of OpenMP on top of Ada to increase its parallel capabilities, enabling the use of unstructured parallelism and advanced parallel heterogeneous architectures.

Although the Ada parallel model provides a simple yet powerful model to exploit structured parallelism in shared memory architectures, the fact that spawning and distribution of work occurs at the same point limits the exploitation of unstructured parallelism, where a task may depend only on some other concurrent tasks. In that respect, OpenMP supports partial synchronizations by means of the `depend` clause, which defines the input and/or output data dependencies existing between tasks. The TDG that honors these dependences is used to drive the execution. As demonstrated in Section 6.3.5, the use of data dependencies can significantly improve performance of parallel Ada programs.

A fundamental requirement of Ada systems is safety. In that regard, OpenMP has been proven to provide the safety requirements imposed by such systems [134]. The main in parallel execution are deadlocks and race conditions. Deadlocks can be palliated using Ada protected objects instead of OpenMP synchronizations. There are sound static analysis techniques [80] if OpenMP mechanisms are still to be used. Race conditions can be solved with concurrency analysis techniques [14], including the automatic discovery of task dependences [131].

Additionally, OpenMP supports an accelerator model featuring the efficient distribution of parallelism in heterogeneous systems the makes the model a firm candidate to be used with Ada, enabling safety-critical systems to efficiently exploit parallel heterogeneous architectures.

6.3.5 Evaluation

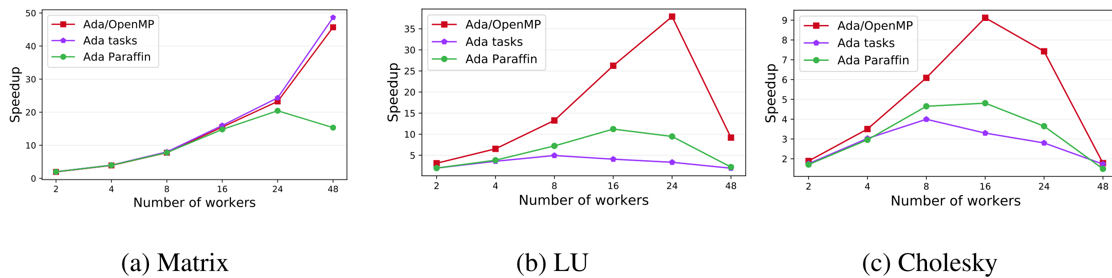
This section shows the evaluation of integrating OpenMP in Ada applications from four different angles: 1) we evaluate the performance benefits of OpenMP compared to other implementations that exploit parallelism in Ada, i.e., native Ada tasks [67] and the Paraffin suite [101]; 2) we evaluate the introduction of raw OpenMP into Ada to exploit fine grain parallelism by means of task dependences; 3) we show that Ada with OpenMP achieves a comparable performance to that of C with OpenMP; and 4) we show the interplay between Ada and OpenMP runtimes.

6.3.5.1 Experimental setup

Runtimes. We use three runtime implementations that support parallelism: 1) the GNU libgomp library for OpenMP from GCC 7.1 [60] 2) the GNAT runtime library for Ada from GCC 7.1 [7], and 3) the Paraffin suit for Ada [101].

Applications. We consider four different applications: 1) a matrix intensive computation resembling image processing algorithms (*Matrix*), 2) a LU factorization (*LU*), 3) a Cholesky decomposition (*Cholesky*), 4) a synthetic application that combines several OpenMP constructs and Ada tasks (*Synthetic*). The Matrix, LU and Cholesky benchmarks, have been parallelized using four different approaches: 1) the Ada parallel model implemented

b]



(a) Matrix (b) LU (c) Cholesky

Figure 6.4: Performance speedup of the Ada parallel programming model implemented with OpenMP, Ada tasks and Paraffin.

with OpenMP (Ada/OpenMP[10]), 2) native Ada tasks, 3) Paraffin and 4) C plus OpenMP (C/OpenMP). The most representative functions of these implementations can be found in Appendix B.2. Additionally, we parallelize the Ada implementation of Cholesky using OpenMP partial synchronizations (task dependences) to demonstrate the benefits of fully integrating OpenMP into Ada by exploiting unstructured parallelism. Finally, Synthetic is used to demonstrate how Ada and OpenMP runtimes coexist by combining OpenMP constructs and Ada tasks managed by the OpenMP and Ada runtimes respectively (OpenMP-parallel constructs are called within Ada tasks).

Platform. We run our experiments in a computing node of the MareNostrum IV supercomputer, which consists of a 2-socket Intel Xeon Platinum 8160 CPU with 24 cores each. The processor operates at 2.10GHz, and features a 33MB L3 cache. (Details of this platform are provided in Section 2.3.1.)

Libraries. We use two instrumentation libraries to analyze the correct interoperability of Ada and OpenMP runtimes: 1) Extrae [17], a tool that gathers information about the performance of parallel applications and generates traces in textual files, and 2) Paraver [21], a performance visualization and analysis tool that uses Extrae traces.

6.3.5.2 Structured parallelism: Ada parallel model, Ada tasks and Paraffin

This section compares the performance speedup of the Ada parallel model (implemented with OpenMP because there is yet no implementation of the Ada tasklet model) with the use of Ada tasks and Paraffin. For such a purpose, we use the Matrix, LU and Cholesky benchmarks. Figure 6.4 shows the speedup obtained for the three benchmarks, considering the three implementations.

In the Matrix example (Figure 6.4a), Ada/OpenMP and Ada tasks produce equivalent speedups. The regular nature of the algorithm can be efficiently mapped to both Ada tasks and OpenMP tasks. On the other hand, Paraffin drops down when the number of workers grows up

[10]Since there is yet no compiler support for Ada using OpenMP directives, and the tasklet model has no implementation either, we simulate this behavior by manually implementing calls to the libgomp runtime library from the source Ada code.

to 48. Note that for Ada and Paraffin, the number of tasks spawned is the same as the number of workers, while in Ada/OpenMP we always use 512 tasks. In LU and Cholesky (Figures 6.4b and 6.4c), Ada/OpenMP clearly outperforms the other implementations because the fine grain synchronization mechanisms provided by OpenMP are more efficient than the manual mapping of parallelism into Ada tasks and the parallelism management performed by Paraffin. In both cases the performance worsens significantly when using the 48 cores because of two reasons: the implementation uses a non-blocked matrix, which introduces overhead in the memory accesses, and 2-socket nature of the machine introduces overhead due to the NUMA effect.

6.3.5.3 Unstructured parallelism: Ada parallel model and OpenMP task dependences

OpenMP allows the definition of partial synchronizations by means of dependence clauses attached to task constructs. This feature allows to further exploit parallelism in highly unstructured parallel applications. As an illustration, we use the Cholesky application from Appendix B.2.1.2 implemented with C and two versions of OpenMP: a) tasks synchronized with taskwaits, and b) tasks synchronized with dependences. Figure 6.5 shows the TDG generated for the version implemented with taskwaits. To generate this graph, we consider all kernels as tasks (*omp_potrf* and *omp_syrk* are not tasks in the original version because they cannot run in parallel with any other task), and we eliminate each taskwait by connecting all its inputs with all its outputs. The figure exhibits that taskwaits are a coarse grain synchronization mechanism that limit the parallelism existing in the application. Figure 6.6 shows the TDG for the version implemented with dependences, which allow exploiting the high level of parallelism (width of the graph) existing in the application.

Figure 6.7 shows the results obtained with the two implementations of Cholesky. The version with dependences outperforms when the number of threads is between 16 and 24, because there are enough resources to exploit the fine grain parallelism existing in the application. Again, the drop in the performance corresponds to a bad memory layout and the NUMA effect of the underlying machine.

6.3.5.4 Performance benefit of OpenMP: Ada vs. C

The OpenMP API efficiently supports the development of parallel applications written in C and Fortran. In this section we prove that OpenMP can be used as well to augment Ada applications with fine grain parallelism, by comparing the performance of three Ada and C codes parallelized with OpenMP: Matrix, LU and Cholesky. Figure 6.8 shows the performance obtained for these three codes implemented with Ada and C, using the same OpenMP parallelization. Ada scales similarly to C in all cases, proving that OpenMP can be used to satisfactorily exploit parallelism in Ada applications. Furthermore, OpenMP reduces the effect of the differences in the underlying languages (C and Ada) when being executed sequentially, delivering a similar execution time for the best parallel versions of both languages[11].

[11] It is not relevant in the context of this thesis to evaluate the differences between C and Ada.

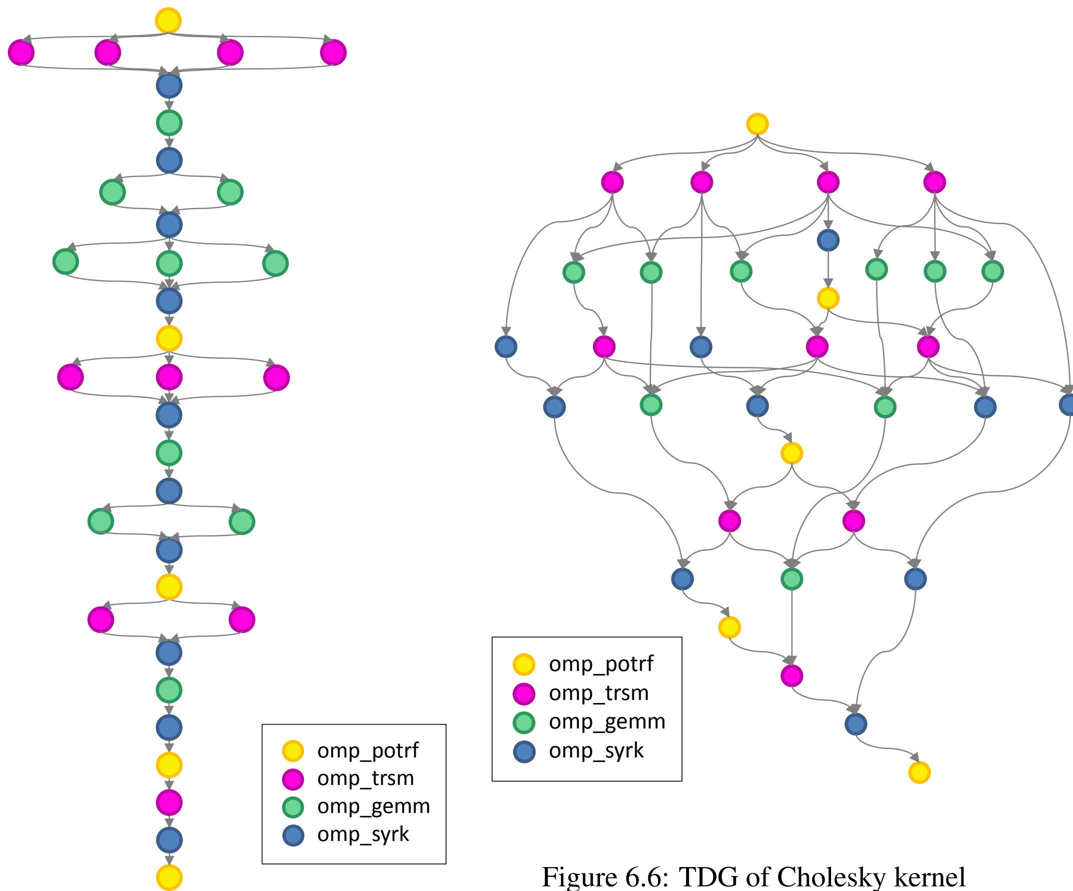


Figure 6.5: TDG of Cholesky kernel implemented using tasks and taskwaits.

Figure 6.6: TDG of Cholesky kernel implemented using tasks with dependences.

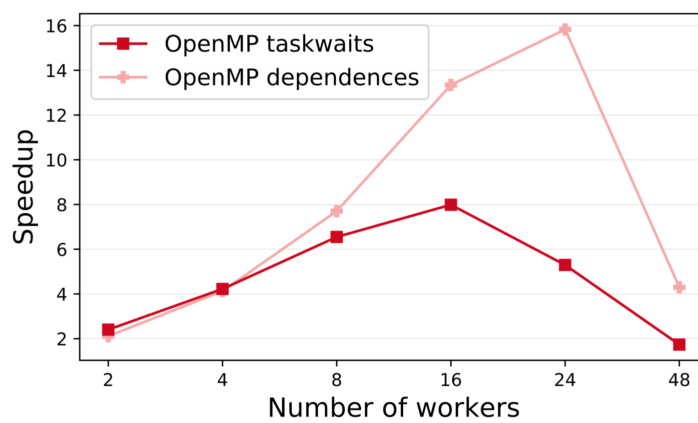
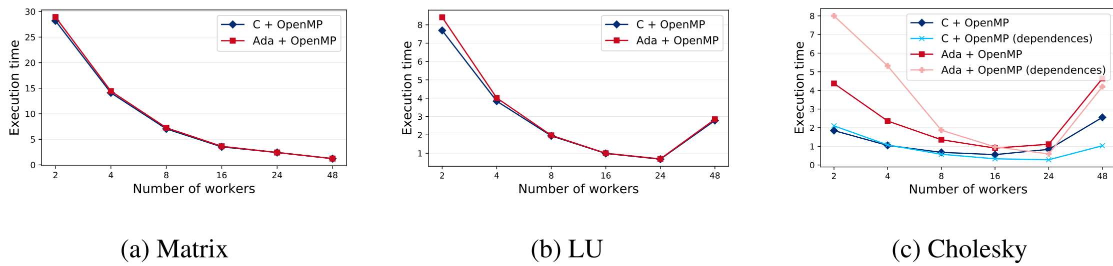


Figure 6.7: Speedup of Ada/OpenMP (structured parallelism) and OpenMP with dependences (unstructured parallelism).



(a) Matrix (b) LU (c) Cholesky

Figure 6.8: Execution time of OpenMP running with Ada and C.

6.3.5.5 Interplay of Ada and OpenMP runtimes

We use a synthetic application to show the coexistence of Ada and OpenMP tasks. The algorithm contains two Ada tasks (one executing periodically every 200ns, and one executing sporadically), and two OpenMP tasks (one performing the intensive computation of Matrix, and one performing light arithmetic computations). OpenMP parallelism is executed within Ada tasks, and the Ada sporadic tasks are released by calling Ada protected objects from within OpenMP tasks.

Figure 6.9 shows a trace of the execution of this algorithm. The x axis represents time, and the y axis represents available workers (labeled *THREADS* in the figure). The horizontal bars contain a unit of execution run in a given period in a given worker, where each color represents a different conceptual unit: the Ada sporadic tasks in yellow (executed in threads 1 and 2), the Ada periodic tasks in turquoise (executed in thread 1), the OpenMP heavy tasks in lilac, and the OpenMP light tasks in pink (the last two executed in all threads). The trace shows how Ada and OpenMP tasks share resources and interplay correctly.

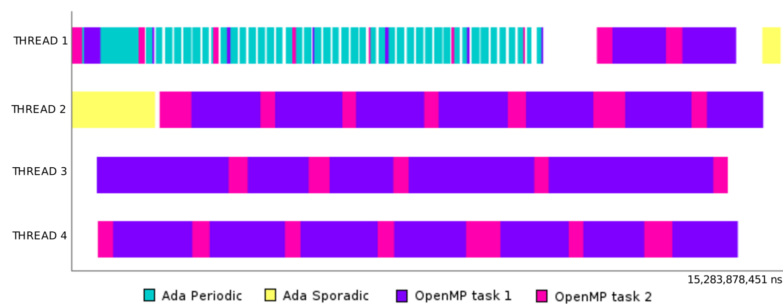


Figure 6.9: Execution trace of the OpenMP and Ada tasks mixed benchmark.

6.3.6 Managing persistent tasks

An important feature in real-time applications is the concept of *persistent* (or *periodic*) task. For example, avionics navigation systems receive periodically multiple input variables of the current flight condition, including air density, throttle lever position, engine temperatures, and engine pressures, among others. These tasks differ from other (*sporadic*) tasks in that they must be executed at regular intervals.

```
task body Periodic is
   T: Time := Clock;
   Period: Time_Span := Milliseconds(200);
begin
   loop
      delay until T;
      -- do work here
      T := T + Period;
   end loop;
end Periodic;
```

Listing 6.11: Example of Ada periodic task using a `delay` statement.

Ada periodic tasks are usually implemented using the `delay` statement for an interval calculated at the end of every period. Listing 6.11 is an example of such behavior.

The current specification of OpenMP does not consider the concept of periodic task, but there have been off-line conversations between members of the OpenMP ARB regarding this matter. One of the achievements of this thesis (further detailed in Section 7.2) is the creation of a discussion group within the ARB in order to tackle real-time aspects. We plan to address this, and other issues, within this group to push their introduction in the specification.

6.3.7 Conclusion

This section takes a step forward in the convergence between the safety-critical and the HPC domains by addressing the integration of OpenMP into Ada. The comparison of the two language specifications reveals that the OpenMP runtime can be used to implement the recently proposed Ada tasklet model, and thus exploit structured fine grain parallelism in Ada applications. Concretely, the OpenMP tasking model, using tied tasks, supports all the preemption model, the progression model and the memory model defined for Ada tasklets.

There are though other implementations that exploit parallelism in Ada, such as Ada tasks and Paraffin. So as to motivate the use of OpenMP to implement tasklets, we compare the three implementations in several benchmarks. The results show the important benefit obtained with OpenMP, mainly because of the efficiency of its fine grain synchronization mechanisms in front of those of Ada tasks and Paraffin. Furthermore, we explore the direct use of OpenMP from Ada to exploit unstructured fine grain parallelism with the use of task dependence clauses. Our results demonstrate the benefits of this kind of partial synchronization against the use of full barrier synchronizations (implemented with the use of taskwaits).

6.4 Correctness for Ada/OpenMP

This section joins the efforts presented in Chapter 4 regarding correctness techniques for OpenMP, and those presented in previous sections of this chapter regarding the adaptation of OpenMP to Ada. Hence, the purpose of this section is to analyze the correctness issues arising from mixed Ada/OpenMP applications, and then define the proper compiler techniques to detect problems in such applications. Particularly, to define an algorithm for detecting data-race conditions in mixed Ada/OpenMP programs.

6.4.1 Related work

In previous sections of this Chapter we have already introduced the different works have already explored the safety requirements necessary for OpenMP to be used in safety-critical environments. On one hand, those that evaluate the time-predictability of OpenMP [96, 138, 139, 145]; on the other hand, those that evaluate the functional safety [132, 134].

This section addresses the functional safety from a compiler perspective. In this regard, previous sections already introduced several compiler analysis techniques to check OpenMP programs for diverse errors, mainly deadlocks [80] and race conditions [27, 93, 133]. However, this techniques do not consider Ada semantics, since they are developed for C, C++ and/or Fortran applications using OpenMP.

The aim of this section is to analyze Ada applications using OpenMP, so we need a unique representation to express the semantics of the two languages. In this respect, different approaches have been used to represent the concurrent semantics of Ada programs: petri nets [48], control flow graphs [51] and different forms of task graphs such as program reachability graphs [124], real-time task digraphs [99] and system dependence nets [158], among others. These representations enable a series of analysis that range from deadlock detection to slicing[12] to complexity measurement.

On another level, most correctness tools for Ada are based on model checking[13], a technique that allows the automatic verification of the correctness of a system. Faria et al. developed ATOS [50], a tool that automatically extracts a SPIN model [63] from an Ada program, as well as a set of desirable properties from a specification annotated by the user in the program, inspired by the SPARK annotation language. Resembling ATOS, GNATprove [123] is a formal verification tool for Ada, based on the GNAT compiler [57] and Meyer's design by contract paradigm [97]. These contracts must be explicitly stated by programmers as preconditions and postconditions for functions and procedures, and loop invariants, all in the syntax of Ada 2012.

6.4.2 Compiler analysis for mixed Ada/OpenMP programs

This section exposes our proposal to solve race conditions in mixed Ada and OpenMP programs. It is structured as follows: first we present the singularities of Ada/OpenMP programs, then we show how we represent Ada/OpenMP programs, next we introduce the algorithm used to detect race conditions in such programs, and finally we show the results of applying the algorithm to a particular test case. For illustration purposes, we use the Ada application *Ravenscar*, defined in Section 7 of the "Ada Ravenscar Profile Guide" [35] as test case. The system modeled in this application includes a periodic process (*Regular_Producer*) that handles offers for a variable amount of workload (*Small_Whetstone*). When the requested workload exceeds a given threshold (*Due_Activation*), the excess load is processed by a sporadic process

[12] *Program slicing* is a technique that consists on reducing a subset of a programs behavior to a minimal form that still produces the same behavior. This technique is used in processes such as program analysis, optimization and debugging.

[13] *Model checking* mechanisms allow exhaustively and automatically the checking of a given model regarding a given specification. Typically, hardware or software components are checked against safety requirements such as the absence of deadlocks and other critical states that can cause a system to crash.

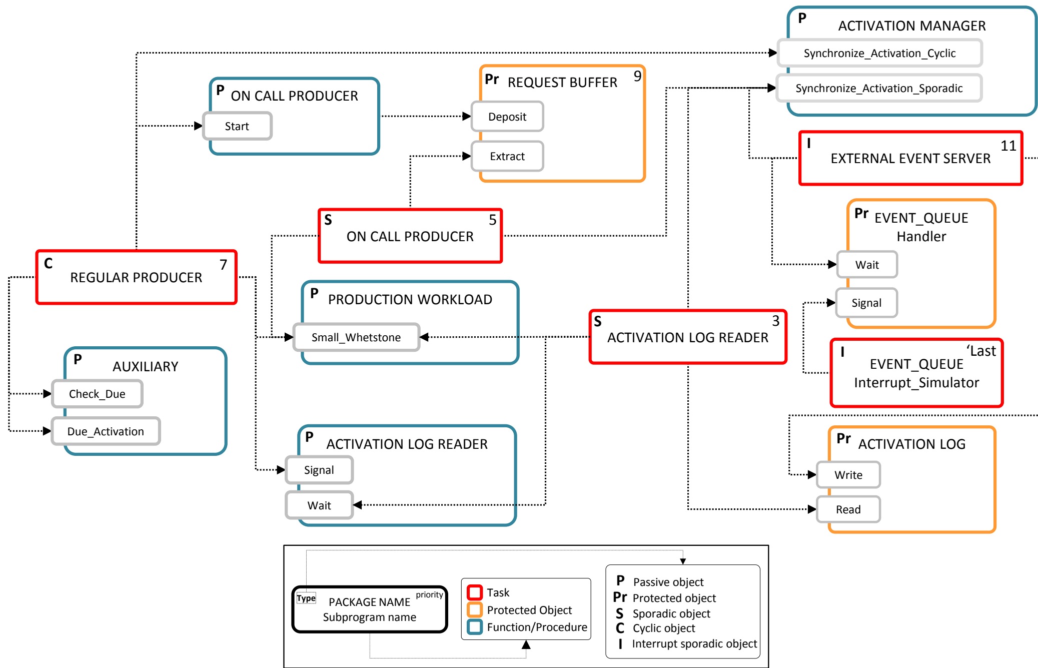


Figure 6.10: HRT-HOOD representation of the *Ravenscar* application defined in Section 7 of the "Ada Ravenscar Profile Guide".

(*On_Call_Producer*). Additionally, interrupts may appear at any point (*External_Event_Server*), and different priorities are used to ensure preference among the different tasks.

Figure 6.10 shows the HRT-HOOD[14] representation of the *Ravenscar* application. There, red boxes represent tasks, blue boxes represent packages with functions and procedures, and yellow boxes represent protected objects with the main entries and procedures.

The *Ravenscar* code illustrates the expressiveness of the Ravenscar profile, for it includes several features of Ada that are of our interest: protected objects, other shared data, synchronous and asynchronous synchronizations, etc. To exemplify how the analysis handles the two levels of parallelism (Ada coarse grain tasks and OpenMP fine grain tasks), we have introduced an OpenMP computation in the *Small_Whetstone* procedure, which turns into the entry point of a sensor fusion operation. Now, the parameter of the function indicates the operation to carry out: 1 means reading sensor A, 2 means reading sensor B, and 3 means fusing the two sensors by adding up its values. The reading of sensor A is performed periodically from *Regular_Producer*, the reading of sensor B is performed sporadically from *On_Call_Producer*, and the fusion is performed sporadically from *Activation_Log_Reader*.

Listing 6.12 uses the syntax proposed in Ada to use OpenMP [134] to illustrate the behavior of the extended OpenMP code. There, the `parallel` construct initiates the parallel execution, although the `single` construct indicates that only one thread will execute the inner statements. Then, the `taskloop` construct indicates that the iterations of the outermost loop are split into chunks that can be executed in parallel using OpenMP tasks.

[14] Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD) is an object-based structured design method

```
1  package body Production_Workload is
2    type Dim is range 1 .. 512;
3    type M is array (Dim, Dim) of Float;
4    M_A, M_B, M_C: M;
5
6    procedure Read_Sensor_A is
7    begin
8      pragma OMP (parallel);
9      pragma OMP (single);
10     pragma OMP (taskloop);
11     for I in Dim loop
12       for J in Dim loop
13         M_A(I,J) := sensor(1, I, J);
14       end loop;
15     end loop;
16   end Read_Sensor_A;
17
18   procedure Read_Sensor_B is
19   begin
20     pragma OMP (parallel);
21     pragma OMP (single);
22     pragma OMP (taskloop);
23     for I in Dim loop
24       for J in Dim loop
25         M_B(I,J) := sensor(2, I, J);
26       end loop;
27     end loop;
28   end Read_Sensor_B;
29
30
31   procedure Fuse_Sensors is
32   begin
33       pragma OMP (parallel);
34       pragma OMP (single);
35       pragma OMP (taskloop);
36       for I in Dim loop
37         for J in Dim loop
38           M_C(I,J) := M_A(I,J)
39                       + M_B(I,J);
40         end loop;
41       end loop;
42   end Fuse_Sensors;
43
44   procedure Small_Whetstone
45             (Workload: Positive) is
46   begin
47     case Workload is
48       when 1 => Read_Sensor_A;
49       when 2 => Read_Sensor_B;
50       when 3 => Fuse_Sensors;
51       when others => null;
52     end case;
53   end Small_Whetstone;
54
55 end Production_Workload;
```

Listing 6.12: OpenMP code inserted in the *Production_Workload* package of the *Ravenscar* application.

6.4.2.1 Concurrency in mixed Ada/OpenMP programs

As introduced previously, pure Ada programs define concurrency by means of tasks, while OpenMP creates parallelism by means of the `parallel` construct, and distributes parallelism by means of worksharing and tasking constructs. When both languages are used together, concurrency may be defined at multiple levels: between Ada tasks, between OpenMP tasks, and between Ada and OpenMP tasks.

Ada protected objects are a robust and lightweight language mechanism for mutual exclusion and data synchronization. For this reason, they are to be used whenever possible to solve race conditions, i.e., when race conditions occur between Ada tasks, between Ada and OpenMP tasks, and between OpenMP tasks that belong to different binding regions (i.e., different parallel regions). This last case is particularly interesting because in C/C++/Fortran OpenMP programs, tasks belonging to different binding regions cannot be concurrent unless there is nested parallelism in the form of nested parallel regions. For that reason, OpenMP does not offer any mechanism to synchronize tasks that belong to different binding regions, except for data synchronizations in the form of flushes. The extra layer of concurrency introduced by Ada opens the door to this kind of situation, and hence, the only mechanism available to synchronize such tasks is the Ada protected object. Finally, to exploit the flexibility of OpenMP, race conditions between OpenMP tasks that belong to the same binding region are to be solved using OpenMP mechanisms: mutual exclusion

for hard real-time systems [33].

<table>
<tr><th colspan="2">Race condition between</th><th>Solution</th></tr>
<tr><td colspan="2">Ada tasks</td><td rowspan="3">Ada mechanisms: protected object</td></tr>
<tr><td colspan="2">Ada and OpenMP tasks</td></tr>
<tr><td rowspan="2">OpenMP tasks</td><td>different binding regions</td></tr>
<tr><td>same binding region</td><td>OpenMP mechanisms:
* Sincronization constructs and clauses:
<code>taskwait</code>, <code>barrier</code>, <code>depend</code>
* Mutual exclusion constructs:
<code>critical</code>, <code>atomic</code>
* Data-sharing attributes:
<code>private</code>, <code>firstprivate</code>, <code>lastprivate</code></td></tr>
</table>

Table 6.1: Solutions for race conditions in an Ada/OpenMP application.

constructs (`atomic` and `critical` constructs), synchronization constructs (e.g., `taskwait` and `barrier`), synchronization clauses (`depend`) and data-sharing clauses (e.g., `private`, `firstprivate` and `lastprivate`). Table 6.1 summarizes our approach to resolve race conditions in each case.

6.4.2.2 Representation of an Ada/OpenMP program

As already pointed out in Section 6.4.1, several representations allow expressing the semantics of an Ada program (e.g., reachability graphs, petri nets, control flow graphs, etc.). However, some representations are not suitable for our purpose, for instance petri nets and reachability graphs, because these express states whereas data flow information is hidden. Furthermore, these representations have other limitations such as the state explosion problem, and the inability of representing recursive programs. Hence, to represent the behavior of an Ada/OpenMP program we use the classic control flow graph (CFG) representation extended to support Ada concurrency and OpenMP parallelism. Our graph draws from the parallel control flow graph presented in Section 3.2.1, and the control flow graph for Ada developed by Fechete et al. [51] and based on the abstract syntax tree (AST) generated by ASIS-for-GNAT [135].

To ease the reading we show the CFGs of the original *Ravenscar* application and the new OpenMP code separately, in Figures 6.11 and 6.12 respectively. The CFG of the original *Ravenscar* code shows the code executed at elaboration time (top of the figure), and the Ada code run during the execution of the program (rest of the figure). Each partial CFG represents a task (*Regular_Producer*, *On_Call_Producer* and *Activation_Log_Reader*). The special nodes *En* and *Ex* express the entry and the exit points of each task, and the OpenMP code is pointed in purple. Finally, the turquoise boxes in the bottom represent some significant shared data, and the edges relating this boxes to the CFG nodes symbolize the type of access to the data: read (dark red) and write (orange).

Regarding the OpenMP code, it is independent from the Ada code because the data structures they manage are different. Note however that the OpenMP parallel tasks are inherently concurrent because they are called from within different Ada tasks, which are in turn concurrent.

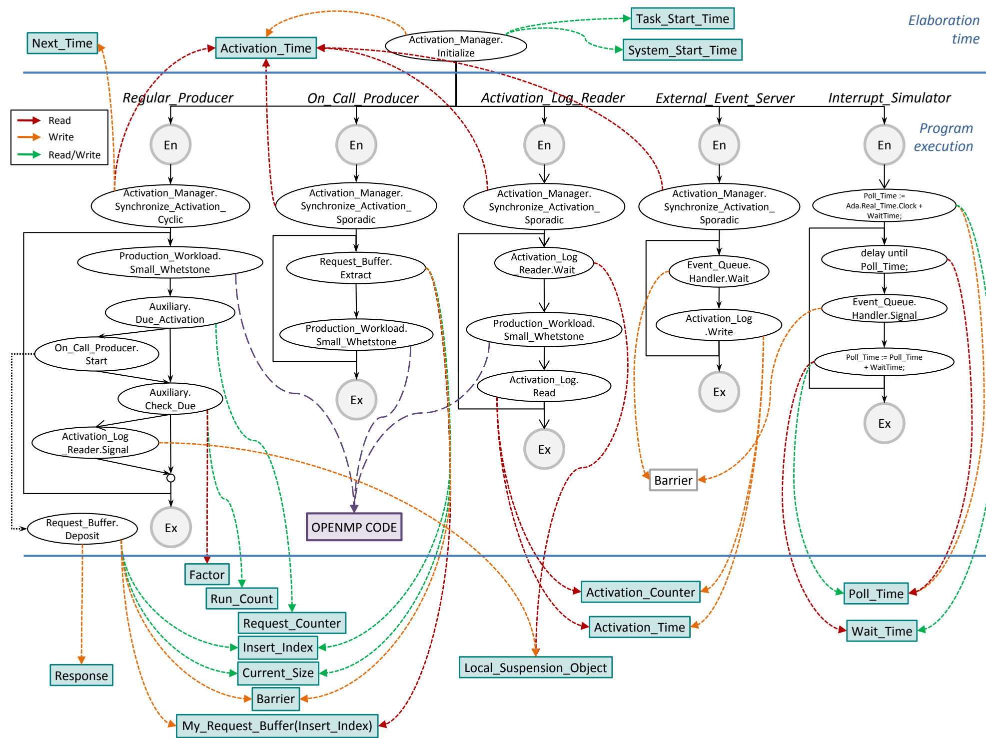


Figure 6.11: Control flow graph of the *Ravenscar* application defined in Section 7 of the "Ada Ravenscar Profile Guide".

Definition 1 *A* block of concurrency*, or* concurrent block*, is a set of portions of code that may execute in parallel.*

Since the application meets the Ravenscar profile, the CFG is particularly simple because all tasks are created at library level, meaning that they start executing at the beginning of the program (after elaboration) and terminate when the program ends (task allocators, task termination and abortion, and task hierarchies, among others, are not allowed). Hence, there are only two blocks of concurrency (split by blue lines in the CFG) that correspond to the code executed during elaboration, and the rest of the code.

6.4.2.3 Correctness analysis

Inspired by the algorithms presented in the scope of OpenMP to automatically determine the data-scoping attributes [129] and the dependence clauses [131] of an OpenMP task, we present an algorithm to find data-race conditions in Ada concurrent programs using OpenMP tasks. Thus, we tackle the race-condition problem from all possible angles: 1) between Ada tasks, 2) between Ada tasks and OpenMP tasks, and 3) between OpenMP tasks. The high-level description of this technique is outlined in Algorithm 9.

Applying the two first steps of the algorithm to our test case results in the CFGs presented in Section 3.2.1. All Ada and OpenMP tasks correspond to the same block of concurrency, hence potential race conditions may occur among all Ada and OpenMP tasks. However, since OpenMP and Ada tasks manage different share data, we can treat them separately.

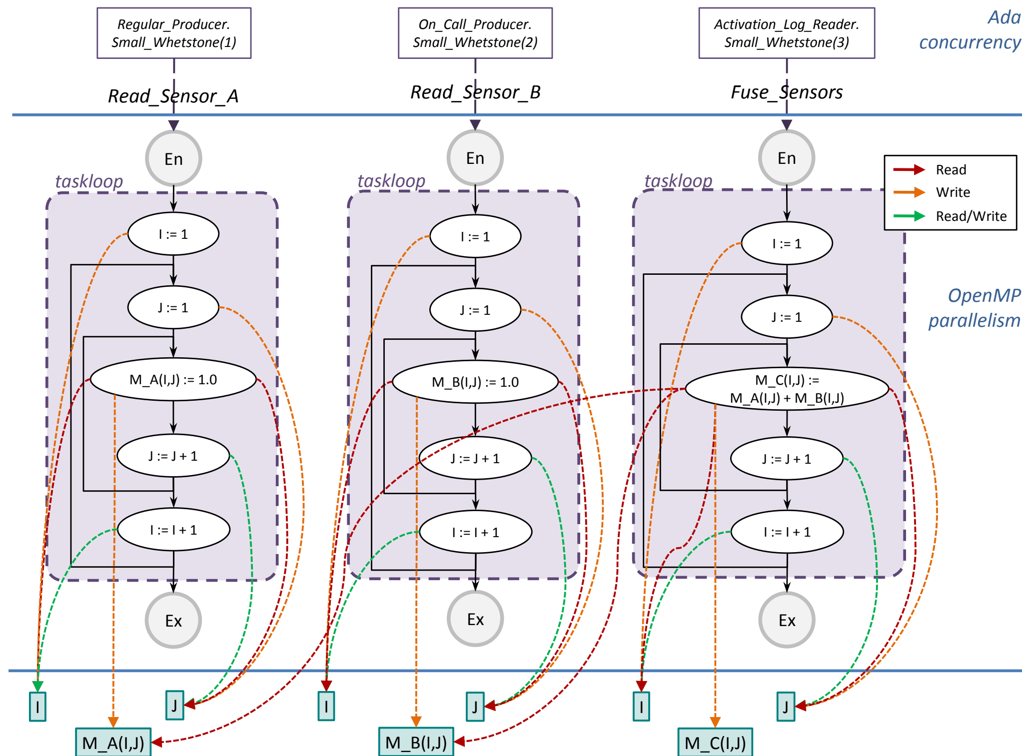


Figure 6.12: Control flow graph of the OpenMP code introduced in the *Small_Whetstone* procedure from the *Ravenscar* application defined in Section 7 of the "Ada Ravenscar Profile Guide".

Applying the third step on the original *Ravenscar* code reveals that accesses to *Activation_Time* are not in a race condition because the write access is in a different concurrent block than the read accesses, the accesses to *My_Request_Buffer(Insert_Index)* are not in a race condition because the object is part of the protected object *Request_Buffer*, and the accesses to *Local_Suspension_Object* are not in a race condition because these operation are atomic with respect to each other, as the standard says. The results of the algorithm on the original *Ravenscar* application found successfully that the code contains no race conditions.

Regarding the analysis of the OpenMP code, the data-sharing rules force a private copy of the induction variable of the taskloop for each thread. Also, the variables declared within the taskloop are private to each thread. As a result, applying the third step of the algorithm on the OpenMP code reveals that accesses to variables *I* and *J* are not in a race condition because they are private to each thread. On the other hand, accesses to the matrices *M_A*, *M_B* and *M_C* are in a race condition because the write access to *M_A* and *M_B* from *Read_Sensor_A* and *Read_Sensor_B* respectively collide with the read access to both variables from *Fuse_Sensor*. The results of the algorithm point to the use of partial synchronizations in the form of task dependence clauses. Task from *Read_Sensor_A* has an output dependency on *M_A(Dim, Dim)*, task from *Read_Sensor_B* has an output dependency on *M_B(Dim, Dim)*, and task from *Fuse_Sensor* has an input dependency on *M_A(Dim, Dim)* and *M_B(Dim, Dim)*, and an output dependency on *M_C(Dim, Dim)*.

Algorithm 9 High-level algorithm to detect race conditions in Ada/OpenMP programs.

1: Build the inter-procedural CFG of the program.
2: Recognize the $block_of_concurrency$ (in a Ravenscar application this is as simple as splitting the elaboration code and the rest of the code).
3: **for each** $block_of_concurrency$ **do**
4: **if** there are concurrent accesses to shared data, and at least one is a write **then**
5: **if** all accesses are within OpenMP tasks that belong to the same binding region **then**
6: **if** the operations are commutative[88] **then**
7: Protect the accesses with an `atomic` or `critical` construct.
8: **else**
9: There are two approaches:
10: * Use full synchronizations: insert a `taskwait` or `barrier` construct between the two accesses.
11: * Use partial synchronizations: follow the algorithm to automatically determine the dependence clauses of an OpenMP tasks.
12: **end if**
13: **else**
14: Propose to wrap the shared data in a protected object.
15: **end if**
16: **end if**
17: **end for**

6.4.2.4 Extending the approach

This work currently assumes a restricted model, where Ada applications follow the Ravenscar profile [35], and considering only the sharing of variables declared in the same scope. This restriction is not related to the approach per se, but instead relate to the complexity of the control flow graph as well as the program code visibility required for the analysis.

In fact, if we consider a more complex concurrency model than Ravenscar, it is necessary to introduce further edges in the graph, as tasks will have other dependencies (e.g., master dependencies, rendezvous, etc.), as well as making the process of determining the concurrency blocks more complex. This will make the analysis more accurate, at the cost of a higher complexity.

Similarly, if full program analysis is available, it will allow to address any data sharing. However, this will introduce further complexity in the approach, as per the complexity of understanding which variables are actually shared. In this context, proposals to cope with this limitation exist for both Ada [147] and OpenMP [132], both consisting in annotations added to APIs of those applications which are to be used as third-party libraries. The Ada annotations include the aspects `Global` and `Potentially_Blocking` to resolve race conditions and deadlocks respectively. The OpenMP annotations, introduced in Section 6.2.3.1, include the directives `globals` and `usage` to resolve race conditions and illegal nesting[15] (including nested regions that can cause deadlocks).

[15]The OpenMP specification (Section 2.17 [113]) defines a series of rules that determine which constructs cannot be nested within each other.

6.4.3 Conclusion

This section enhances the use of OpenMP in safety-critical environments by addressing the safety of Ada programs in the presence of parallel computation implemented with OpenMP. Hence, it provides one step further in the work presented in the current chapter to enable OpenMP fine-grained parallelism in Ada. With this purpose, the section introduces a new compiler analysis technique that can identify potential race conditions in Ada, both considering Ada tasks and parallel OpenMP code. The technique is built on top of three components: a) a graph representation that includes both control- and data-flow dependencies of concurrent and parallel code, b) an adaptation of existent compiler techniques developed for sequential languages to consider Ada tasks, and c) compiler methods that detect data races and guide the programmer in solving them.

6.5 Impact

On one hand, the works regarding the adaptation of OpenMP to the safety critical domain are being taken into account by the OpenMP ARB. Currently, a mailing list has started with the aim of defining the topics to be discussed regarding safety from two different perspectives: functionality and time-predictability. These discussions shall serve as the basis for determining how safety issues must be tackled within the structure of committee.

On the other hand, the works regarding the introduction of OpenMP into Ada have made an impression on the Ada community and are going to be discussed in the next IRTAW, to be hold on April 2018. This will represent another step in the introduction of fine grain parallelism in Ada, a topic that has already been under discussion in previous editions of the workshop, where only the tasklet model was under consideration.

6.6 Conclusion

This chapter aims to widen the use of OpenMP in new areas, particularly the safety-critical real-time domain. We tackle this issue from two different angles: 1) how the OpenMP specification can be adapted to meet the requirements of safety-critical systems, and 2) how safe languages can benefit from OpenMP. Regarding the former, OpenMP shall be analyzed in two directions: functional safety and time predictability. Regarding the latter, three aspects must be considered: language syntax, compiler analysis and runtime support. Figure 6.13 shows an schema of this scenario, and displays as well the status of our work regarding each of these angles.

Regarding OpenMP, we tackle functional safety in the specification, and let time-predictability out of the scope of this thesis[16]. In this context, we provide prove that the OpenMP specification has few features that can jeopardize the safety of a system. Particularly, we detect features that could be solved at compile-time and/or run-time, and provide solutions for them. A small subset

[16]The analysis of the timing properties of OpenMP has been addressed in other works [96, 139, 145, 155] cited along the sections of this chapter.

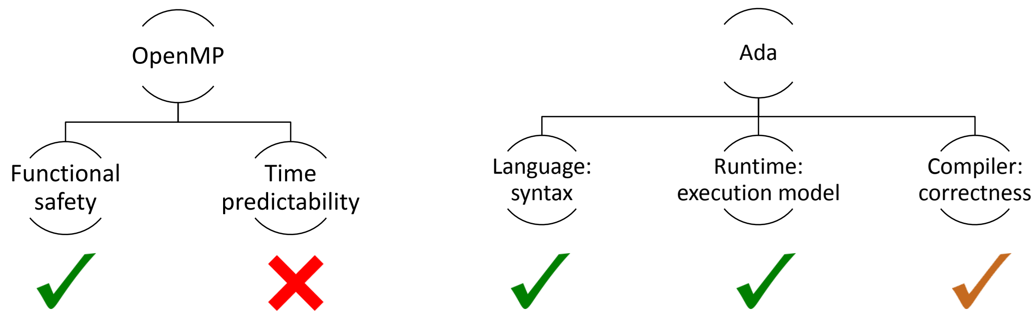


Figure 6.13: Schema of the issues addressed in the context of applying OpenMP in the safety-critical domain by means of Ada.

of features is identified as not analyzable, and we propose to restrict the use of these features depending on the level of criticality of the system. Furthermore, we propose the use of two new directives to enable whole program analysis even when third-party libraries are used.

Furthermore, we address the runtime support for integrating OpenMP into Ada, and analyze the OpenMP and the Ada tasklet models proving that both models have equivalent features, hence the same safety properties. This proves that OpenMP can be used to implement the tasklet model to exploit structured fine-grain parallelism. Additionally, we show how Ada can further exploit the OpenMP tasking model and its partial synchronizations to implement unstructured parallelism.

Finally, we cover the compiler support regarding static analysis by designing a new analysis technique that allows discovering data-race conditions in Ada codes including Ada tasks and/or OpenMP tasks. The algorithm is sound and is analyzed using a well known Ravenscar example. The model used to represent the semantics of an Ada/OpenMP program, the CFG, is trivial for a Ravenscar application. It remains as a future work the extension of this representation to cover non-Ravenscar features such as dynamic task allocation or task abortion among others.

Regarding Ada, we tackle the problem at all possible levels. First, we propose a new syntax for OpenMP adapted to that of Ada to keep its safety properties. Second, we prove the equivalence between the OpenMP execution model and that of Ada tasklets. Hence, we show evidence of how OpenMP can be used to implement the tasklet model, and how OpenMP can be used directly in Ada codes to exploit its fine-grain parallel capabilities. Third, we propose a new compiler technique able to detect data-race conditions in Ada codes using or not Ada tasks and OpenMP tasks. In this sense, it remains as future work to test this technique with non-Ravenscar applications, and also to extend it to detect not only race conditions but also deadlocks and other issues related to parallel execution.

7

Discussion

7.1 Conclusion

In this thesis, we present different contributions that tackle compiler analysis within the scope of the OpenMP programming model from three different perspectives: compiler algorithms, language constructions and runtime algorithms. These contributions, thoroughly described in Section 1.3, include: 1) compiler analysis techniques dedicated to determine OpenMP's functional safety, 2) compiler techniques dedicated to enlighten OpenMP runtimes, 3) the study of the suitability of OpenMP for the safety-critical domain, 4) the integration of OpenMP into Ada, and 5) the development of compiler analysis techniques tacking Ada programs parallelized with OpenMP.

Generally speaking, we conclude that there is a need to include methodologies for determining functional safety in OpenMP. Programmers may take profit of correctness mechanisms to avoid undesired results at both compile-time and run-time. In this context, solutions may come from different angles. On one hand, the OpenMP specification could introduce new features to help detecting correctness issues. In that regard, we propose new directives that enable whole program analysis even when third-party libraries are used. We also note the need to introduce error handling techniques in the specification. On the other hand, we describe a variety of analyses that should be implemented in *safety*-OpenMP compliant compilers to detect situations that may entail a hazard regarding the correctness of the program. Among these, we highlight the necessity of mechanisms for detecting race-conditions and deadlocks. Furthermore, we present different techniques, specifically designed for OpenMP tasks, that allow detecting not only races and deadlocks, but also incongruities regarding the use of OpenMP tasks data-sharing attributes and dependences. Finally, we propose some changes to be applied in OpenMP runtimes to cover those cases that cannot be tackled at the specification level, nor in the compiler.

The previously mentioned mechanisms lead us to a scenario where OpenMP could easily be applied to environments that are currently beyond its reach, as it currently is the safety-critical domain. In that sense, we conclude that functional safety can be ensured in OpenMP programs if the methodologies we propose are introduced at all levels (programming language, compiler and runtime). As to materialize the use of OpenMP in safety-critical systems, we integrate OpenMP and Ada and demonstrate the usefulness of OpenMP in boosting the performance of

Ada applications. In this context, the complete integration of OpenMP in safety-critical domains still requires an accurate study of OpenMP regarding response-time analysis.

Also in the direction of using OpenMP in new environments, we propose a compiler technique to statically generate a TDG. This graph guides the scheduling of OpenMP tasks by considerably reducing the memory usage because only a very light mechanism is needed at runtime to decide whether a task is dependent on other tasks or not. This feature is very useful in systems where memory constraints are tight, such as embedded systems.

7.2 Impact

Several works in this thesis have had and are having an impact in the international community. This section introduces the repercussion of this work in three different facets: 1) the European projects where this work has been, is being or will be used, 2) the influence on the evolution of different programming models, and 3) other Master's and PhD thesis where this work has a significant impact.

7.2.1 European projects

P-SOCRATES, Parallel Software Framework for Time-Critical Many-core Systems

P-SOCRATES is a European project that lasted between 2013 and 2016 [121, 150]. The goal of the project was to develop a complete framework, from the conceptual design to the physical implementation, to combine real-time embedded mapping and scheduling techniques with high-performance parallel programming models and associated tools.

The static generation of a task dependency graph based on an OpenMP/OmpSs application is integrated as part of the software stack of P-SOCRATES, and enables two important fields of study within the project:

1. The application of the sporadic-DAG scheduling model, a well-known technique in scheduling theory to represent real-time systems, upon which schedulability guarantees can be derived.
2. The development of task-to-thread work-conserving mapping strategies based on the OpenMP tied and untied tasking models: breadth-first scheduling, work-first scheduling, fully static mapping and limited preemption scheduling.

AXIOM, Agile, eXtensible, fast I/O Module for the cyber-physical era

AXIOM is a 3-year European project that started in 2015 [12, 151]. It aims at researching new software/hardware architectures for the future Cyber-Physical Systems (CPSs). These systems are expected to react in real-time, provide enough computational power for the assigned tasks consuming the least possible energy, scale up through modularity and allow for an easy programmability across performance scaling.

The static generation of a task dependency graph for OpenMP/OmpSs applications is being used with two main purposes:

1. The study and implementation of new scheduling policies for OmpSs clusters . The use of static TDGs enables the estimation and exploration of different scheduling policies by

assigning weights representing communication costs, data copies, execution costs, etc. to the nodes and edges of the graph. Furthermore, the static TDGs may help to reduce the cost of creating tasks and controlling the dependences because this may be done off-line.

2. The implementation of a code generator from OmpSs code for the data-flow based *xsmll* (eXtended Shared Memory Low Level specification) runtime [54]. In the xsmll model, threads executing tasks must receive a descriptor of its successors. Hence, the whole task dependency graph must be expanded prior to the generation of the code.

CLASS, Edge and CLoud Computation: A Highly Distributed Software Architecture for BigData AnalyticS

CLASS is a 3-year European project that started in January 2018. The main objective of the project is to develop a novel software architecture to help big data developers to fully benefit from a combined data-in-motion and data-at-rest analysis by efficiently distributing data and process mining along the *compute continuum* (from edge to cloud resources) in a complete and transparent way, while providing sound real-time guarantees imposed by autonomous vehicles.

The static generation of a TDG based on an OpenMP/OmpSs application is to be used within the software stack to be developed in the project in order to help in the scheduling of different tasks among the compute continuum. The use of the static TDG will be enclosed in the frame of the COMPSs programming model [90], which already uses a dynamically generated task dependency graph equivalent to that of OpenMP and OmpSs.

7.2.2 Programming models

The works that have been conducted in the scope of this thesis related to the convergence of the HPC and the real-time domains are having a significant impact in two communities:

- the OpenMP language committee. Our work has motivated the creation of a discussion group within the OpenMP ARB in order to tackle real-time aspects in the OpenMP specification. This group is currently defining the topics to be discussed and the members that want to participate in the discussion.
- the Ada language committee. In the last years, the committee has been considering the expansion of the core language to support fine grain parallelism. OpenMP is now another option under consideration that may avoid the burden of developing new syntax for that purpose, such as the tasklet model presented in Section 6.4, as well as bring in the fore many interesting features.

7.2.3 Other thesis

The analysis platform developed during this PhD has been and is being used by other students to carry out their own thesis. Next we show the list of Degree, Master's and PhD thesis that take profit from our framework:

- Diego Caballero. *SIMD@OpenMP: A Programming Model Approach to Leverage SIMD Features*. Ph.D. programme by ”Computer Architecture”, Final Dissertation. Department of Computer Architecture, Universitat Politècnica de Catalunya. November, 2015.
- Maria A. Serrano. *Time-predictable parallel programming models*. Ph.D. programme by ”Computer Architecture”, Final Dissertation. Department of Computer Architecture Universitat Politècnica de Catalunya. To be presented in 2019.
- Daniel Peyrolon. *Code generation for the dataflow-based xsmll runtime*. Master in Innovation and Research in Informatics, Final Thesis. Facultat d'Informàtica de Barcelona. October, 2017.
- Juan López. *Task scheduling in a disjoint global memory model for FPGA accelerated clusters*. Bachelor Degree in Informatics Engineering. Final Degree Project. To be presented in 2018.

7.3 Future work

The analysis infrastructure build on the Mercurium compiler as part of this thesis lays a solid basis for further research on analysis techniques focused on OpenMP. This research can include new compiler analysis algorithms to enhance the accuracy of the results obtained with our correctness analysis tool. In this regard, we are currently working on the extension of range analysis techniques [125] to support OpenMP, which could be used to accurately determine overflows in the use of task dependence expressions. Along the same lines, new analysis algorithms can address different correctness issues regarding the use of OpenMP tasks.

In the context of the generation of the TDG at compile-time, we are working on different directions. On one hand, we plan to augment the information collected around the TDG to enable the study of static scheduling techniques. Concretely, we plan to use the previously mentioned range analysis to argue about the data consumed and produced in OpenMP tasks, and also we will extract other information such as tasks weight, communications cost, etc., to elaborate new static scheduling strategies. Furthermore, we want to use the static TDG generation to enhance productivity in high-performance domains by preallocating data from the compiler, and prefetching data from the runtime scheduler.

Regarding the integration of OpenMP in the safety-critical domain, there are different issues to consider. On one hand, we develop about the suitability of OpenMP regarding functional safety. In this regard, there is still a need to analyze the OpenMP specification in terms of time-predictability. On the other hand, the Ada and OpenMP runtimes still have to be integrated, and the issues raised from this integration need to be addressed. Furthermore, are currently working on the adaptation of the algorithms developed for automatically scoping variables and for automatically determining the dependence clauses in task constructs, in order to adapt them to both the Ada concurrent model and the Ada parallel model.

7.4 Publications

The work conducted in this thesis resulted in six publications, which are listed as follows:

- Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. *Compiler analysis for OpenMP tasks correctness*. Proceedings of the 12th ACM International Conference on Computing Frontiers, CF. Ischia, Italy. May 18-21, 2015.
- Roberto E. Vargas, Sara Royuela, Maria A. Serrano, Eduardo Quiñones, and Xavi Martorell. *A Lightweight OpenMP4 Run-time for Embedded Systems*. 21st Asia and South Pacific Design Automation Conference, ASP-DAC. Macau, China. January 25-28, 2016.
- Sara Royuela, Xavier Martorell, Eduardo Quiñones, and Luis Miguel Pinho. *OpenMP Tasking Model for Ada: Safety and Correctness*. 22nd International Conference on Reliable Software Technologies, Ada-Europe. Vienna, Austria. June 12-16, 2017.
- Sara Royuela, Alejandro Duran, Maria A. Serrano, Eduardo Quiñones, and Xavier Martorell. *Functional Safety for Hard Real-Time OpenMP*. Proceedings of the 13th International Workshop on OpenMP, IWOMP. Stony Brook, New York, USA. September 21-22, 2017.
- Sara Royuela, Eduardo Quiñones, and Luis Miguel Pinho. *Converging Safety and High-performance Domains: Integrating OpenMP into Ada*. Proceedings of the 21st Conference on Design, Automation and Test in Europe, DATE. Dresden, Germany. March 19-23, 2018.
- Sara Royuela, Xavier Martorell, Luis Miguel Pinho and Eduardo Quiñones. *Safe Parallelism: Compiler Analysis Techniques for Ada and OpenMP*. 23rd International Conference on Reliable Software Technologies, Ada-Europe. Lisbon, Portugal. June 18-22, 2018.

Additionally, some of the work presented in this thesis was used in the following publication:

- Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. *Optimizing overlapped Memory Accesses in User-directed vectorization*. Proceedings of the 29th ACM on International Conference on Supercomputing. Newport Beach, California, USA. June 8-11, 2015.

Furthermore, as part of the future work, there are three ongoing publications listed as follows:

- Maria A. Serrano, Sara Royuela, and Eduardo Quiñones. *Chapter 3 - Predictable Parallel Programming with OpenMP* from the book *High-performance and Time-predictable Embedded Computing*. River Publishers, 2018.
- Luis Miguel Pinho, Eduardo Quiñones and Sara Royuela. *Position paper: combining the tasklet model with OpenMP*. 19th International Real-Time Ada Workshop. Benicàssim, Spain. April 18-20, 2018.
- Sara Royuela, Luis Miguel Pinho and Eduardo Quiñones. *Solving race conditions in Ada OpenMP parallel programs*. International Symposium on Code Generation and Optimization. Vienna, Austria. 2019.

Finally, it is also worth to mention the publications presented in the context of the master's thesis because they are the basis of many of the work presented in this thesis:

- Roger Ferrer, Sara Royuela, Diego Caballero, Alejandro Duran, Xavier Martorell, and Eduard Ayguade. *Mercurium: Design Decisions for a S2S Compiler*. Cetus Users and Compiler

Infrastructure Workshop in conjunction with PACT. Galveston Island, Texas, USA. October 10, 2011.

- Sara Royuela, Alejandro Duran, Chunhua Liao, and Daniel J. Quinlan. *Auto-scoping for OpenMP Tasks*. Proceedings of the 8th International Workshop on OpenMP, IWOMP. Rome, Italy. June 11-13, 2012.
- Sara Royuela, Alejandro Duran, and Xavier Martorell. *Compiler automatic discovery of OmpSs task dependencies*. International Workshop on Languages and Compilers for Parallel Computing, LCPC. Tokyo, Japan. September 11-13, 2012.

Bibliography

[1] Ada Conformity Assessment Authority. *Rationale for Ada 2012: Multiprocessors*. 2014. URL: {{http://www.ada-auth.org/standards/12rat/html/Rat12-5-3.html}}.

[2] Ada-Europe. *Ada Reference Manual ISO/IEC 8652:2012(E) – Timed Entry Calls*. 2012. URL: {{http://www.adaic.org/resources/add_content/standards/12rm/html/RM-9-7-2.html}}.

[3] Ada-Europe. *Ada Reference Manual ISO/IEC 8652:2012(E) – Annex D: Real-Time Systems*. 2013. URL: {{http://www.adaic.org/resources/add_content/standards/12rm/html/RM-D.html}}.

[4] Ada-Europe. *Ada Reference Manual ISO/IEC 8652:2012(E) – Conditional Entry Calls*. 2013. URL: {{http://www.adaic.org/resources/add_content/standards/12rm/html/RM-9-7-3.html}}.

[5] Ada-Europe. *Ada Reference Manual ISO/IEC 8652:2012(E) – Entry Queuing Policies*. 2013. URL: {{http://www.adaic.org/resources/add_content/standards/12rm/html/RM-D-4.html}}.

[6] Ada-Europe. *Ada Reference Manual ISO/IEC 8652:2012(E) – Task Priorities*. 2013. URL: {{http://www.adaic.org/resources/add_content/standards/12rm/html/RM-D-1.html}}.

[7] AdaCore. *GNAT User's Guide for Native Platform*. 2017. URL: {https://gcc.gnu.org/onlinedocs/gnat_ugn.pdf}.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley Boston, 1986.

[9] S. Aldea López. "Compile-time support for thread-level speculation". PhD thesis. Universidad de Valladolid, 2014.

[10] S. Arandi, G. Michael, P. Evripidou, and C. Kyriacou. "Combining compile and run-time dependency resolution in data-driven multithreading". In: *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*. IEEE. 2011, pp. 45–52.

[11] Arm. *Cortex-A57*. 2017. URL: {https://developer.arm.com/products/processors/cortex-a/cortex-a57}.

[12] AXIOM. *Agile, eXtensible, fast I/O Module for the cyber-physical era*. 2015. URL: http://www.axiom-project.eu.

[13] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. "Parallelizing dense and banded linear algebra libraries using SMPSs". In: *Concurrency and Computation: Practice and Experience* 21.18 (2009), pp. 2438–2456.

[14] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. "A theory of data race detection". In: *Parallel and distributed systems: testing and debugging*. ACM. 2006, pp. 69–78.

[15] Barcelona Supercomputing Center. *OmpSs User Guide*. URL: {https://pm.bsc.es/ompss-docs/user-guide}.

[16] Barcelona Supercomputing Center. *Course on programming models using OmpSs*. 2014. URL: {https://eventos.redclara.net/indico/event/311/overview}.

[17] Barcelona Supercomputing Center. *Extrae*. 2017. URL: {https://tools.bsc.es/extrae}.

[18] Barcelona Supercomputing Center. *MareNostrum III User's Guide*. 2017. URL: {https://www.bsc.es/support/MareNostrum3-ug.pdf}.

[19] Barcelona Supercomputing Center. *MareNostrum IV User's Guide*. 2017. URL: {https://www.bsc.es/support/MareNostrum4-ug.pdf}.

[20] Barcelona Supercomputing Center. *OmpSs specification*. 2017. URL: {https://pm.bsc.es/ompss-docs/spec/index.html}.

[21] Barcelona Supercomputing Center. *Paraver*. 2017. URL: {https://tools.bsc.es/paraver}.

[22] Barcelona Supercomputing Center. *Programming Models group at BSC*. 2017. URL: {https://pm.bsc.es}.

[23] J. Barnes. *Safe and secure software: An invitation to Ada 2012*. AdaCore, 2015, pp. 107–126. URL: {http://www.adacore.com/uploads/technical-papers/SafeSecureAdav2015-covered.pdf}.

[24] S. Baruah. "Improved multiprocessor global schedulability analysis of sporadic dag task systems". In: *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE. 2014, pp. 97–105.

[25] N. Bascelija. "Sequential and Parallel Algorithms for Cholesky Factorization of Sparse Matrices". In: *WSEAS: Mathematical Applications in Science and Mechanics* (2013).

[26] A. Basumallik and R. Eigenmann. "Incorporation of OpenMP memory consistency into conventional dataflow analysis". In: *International Workshop on OpenMP*. Springer. 2008, pp. 71–82.

[27] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. "ompVerify: polyhedral analysis for the OpenMP programmer". In: *International Workshop on OpenMP*. Springer. 2011, pp. 37–53.

[28] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. “P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator”. In: *Conference on Design, Automation & Test in Europe*. EDA Consortium. 2012, 983–987.

[29] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. “Cilk: An efficient multithreaded runtime system”. In: *Journal of parallel and distributed computing* 37.1 (1996), pp. 55–69.

[30] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. “DAGuE: A generic distributed DAG engine for high performance computing”. In: *Parallel Computing* 38.1 (2012), pp. 37–51.

[31] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. “Productive cluster programming with OmpSs”. In: *European Conference on Parallel Processing*. Springer. 2011, pp. 555–566.

[32] P. Burgio, G. Tagliavini, A. Marongiu, and L. Benini. “Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters”. In: *Conference on Design, Automation & Test in Europe*. EDA Consortium. 2013, 1504–1509.

[33] A. Burns and A. J. Wellings. “HRT-HOOD: A structured design method for hard real-time systems”. In: *Real-Time Systems* 6.1 (1994), pp. 73–114.

[34] A. Burns, B. Dobbing, and G. Romanski. “The Ravenscar tasking profile for high integrity real-time programs”. In: *International Conference on Reliable Software Technologies*. Springer. 1998, pp. 263–275.

[35] A. Burns, B. Dobbing, and T. Vardanega. “Guide for the use of the Ada Ravenscar Profile in high integrity systems”. In: *ACM SIGAda Ada Letters* 24.2 (2004), pp. 1–74.

[36] D. Caballero. “SIMD@OpenMP: a programming model approach to leverage SIMD features”. PhD thesis. Universitat Politècnica de Catalunya · BarcelonaTech (UPC), 2015.

[37] D. Caballero, S. Royuela, R. Ferrer, A. Duran, and X. Martorell. “Optimizing Overlapped Memory Accesses in User-directed Vectorization”. In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. Newport Beach, California, USA: ACM, 2015, pp. 393–404.

[38] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. “Implementing OpenMP on a high performance embedded multicore MPSoC”. In: *International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–8.

[39] J. Cownie and S. Moore. “Portable OpenMP debugging with totalview”. In: *European Workshop on OpenMP*. 2000.

[40] N. CUDA^{TM}. *Nvidia CUDA C Programming Guide*. 2012. URL: {{http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf}}.

[41] B. D. De Dinechin, D. Van Amstel, M. Poulhiès, and G. Lager. “Time-critical computing on a single-chip massively parallel processor”. In: *Conference on Design, Automation & Test in Europe*. IEEE. 2014, pp. 1–6.

[42] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel. “A distributed run-time environment for the kalray mppa®-256 integrated manycore processor”. In: *Procedia Computer Science* 18 (2013), pp. 1654–1663.

[43] R. DO. “178C”. In: *Software considerations in airborne systems and equipment certification* (2011).

[44] R. DO. “178C”. In: *Software considerations in airborne systems and equipment certification* (2011).

[45] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. “Ompss: a proposal for programming heterogeneous multi-core architectures”. In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193.

[46] A. Duran, R. Ferrer, J. J. Costa, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. “A proposal for error handling in OpenMP”. In: *International Journal of Parallel Programming* 35.4 (2007), pp. 393–416.

[47] M. Eslamimehr and J. Palsberg. “Sherlock: scalable deadlock detection for concurrent programs”. In: *International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 353–365.

[48] S. Evangelista, C. Kaiser, J.-F. Pradat-Peyre, and P. Rousseau. “Quasar: a new tool for concurrent Ada programs analysis”. In: *Ada-Europe*. Springer. 2003, pp. 168–181.

[49] X. Fan, M. Mehrabi, O. Sinnen, and N. Giacaman. “Exception Handling with OpenMP in Object-Oriented Languages”. In: *International Workshop on OpenMP*. Springer. 2015, pp. 115–129.

[50] J. M. Faria, J. Martins, and J. S. Pinto. “An Approach to Model Checking Ada Programs”. In: *17th Ada-Europe International Conference on Reliable Software Technologies*. Ed. by M. Brorsson and L. M. Pinho. Stockholm, Sweden: Springer, 2012, pp. 105–118.

[51] R. Fechete and G. Kienesberger. “A Framework for CFG-Based Static Program Analysis of Ada Programs”. In: *13th Ada-Europe International Conference on Reliable Software Technologies*. Ed. by F. Kordon and T. Vardanega. Venice, Italy: Springer, 2008, pp. 130–143.

[52] R. Ferrer, S. Royuela, D. Caballero, A. Duran, X. Martorell, and E. Ayguadé. “Mercurium: Design decisions for a s2s compiler”. In: 2011.

[53] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu. “A real-time scheduling service for parallel tasks”. In: *19th Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2013, pp. 261–272.

[54] R. Giorgi, P. Gai, B. Morelli, and S. Garzarella. *D7.1 – Initial AXIOM Evaluation Platform (AEP) definition and initial tests*. Tech. rep. 2016. URL: http://www.axiom-project.eu/wp-content/uploads/2015/04/AXIOM-D71-v1.pdf.

[55] R. Giorgi et al. “TERAFLUX: Harnessing dataflow in next generation teradevices”. In: *Microprocessors and Microsystems* 38.8 (2014), pp. 976–990.

[56] GNU. *GCC 5.4 manuals*. 2016. URL: {https://gcc.gnu.org/onlinedocs/5.4.0}.

[57] GNU. *GNAT*. https://www.gnu.org/software/gnat. 2016.

[58] GNU. *Offloading Support in GCC*. 2016. URL: {https://gcc.gnu.org/wiki/Offloading}.

[59] GNU. *Link Time Optimization*. 2017. URL: {https://gcc.gnu.org/onlinedocs/gccint/LTO.html}.

[60] GNU. *The GOMP project*. 2017. URL: {https://gcc.gnu.org/projects/gomp}.

[61] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun. “On-the-fly detection of data races in OpenMP programs”. In: *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. ACM. 2012, pp. 1–10.

[62] T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, and T. Boku. “Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP”. In: *International Workshop on OpenMP*. Springer. 2009, pp. 15–27.

[63] G. J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[64] L. Huang, D. Eachempati, M. W. Hervey, and B. Chapman. “Extending global optimizations in the OpenUH compiler for OpenMP”. In: *Open64 Workshop at CGO*. Citeseer. 2008.

[65] L. Huang, G. Sethuraman, and B. Chapman. “Parallel data flow analysis for openmp programs”. In: *International Workshop on OpenMP*. Springer. 2007, pp. 138–142.

[66] IBM. *IBM Parallel Environment*. 2016. URL: {http://www-03.ibm.com/systems/power/software/parallel}.

[67] IEC. “8652: 2012 Programming Languages and their Environments–Programming Language Ada”. In: *International Standards Organization* ().

[68] Infineon. *AURIX™– Safety joins Performance*. 2017. URL: {https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/channel.html?channel=db3a30433727a44301372b2eefbb48d9}.

[69] Intel®. *OpenMP Runtime Library*. 2016. URL: {https://www.openmprtl.org}.

[70] Intel® Corporation. *Interprocedural Optimization*. 2017. URL: {https://software.intel.com/en-us/node/522666}.

[71] International Electrotechnical Comission. *IEC 61508 Edition 2.0, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. 2009.

[72] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*. 2009.

[73] ISO/IEC JTC 1/SC 22. *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. 2011. URL: {https://www.iso.org/standard/50372.html}.

[74] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy. "Helgrind+: An efficient dynamic race detector". In: *International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–13.

[75] R. Kaiser and S. Wagner. "Evolution of the PikeOS microkernel". In: *First International Workshop on Microkernels for Embedded Systems*. 2007.

[76] P. Kegel, M. Schellmann, and S. Gorlatch. "Using OpenMP vs. Threading Building Blocks for medical imaging on multi-cores". In: *Euro-Par Conference on Parallel Processing* (2009), pp. 654–665.

[77] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[78] A. J. Kornecki. *Software Development Tools for Safety-Critical, Real-Time Systems Handbook*. Office of Aviation Research and Development, Federal Aviation Administration, 2007.

[79] G. Krawezik. "Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors". In: *ACM symposium on Parallel algorithms and architectures*. ACM. 2003, pp. 118–127.

[80] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. "Sound static deadlock analysis for C/Pthreads". In: *International Conference on Automated Software Engineering*. IEEE. 2016, pp. 379–390.

[81] B. Kuhn, P. Petersen, and E. O'Toole. "OpenMP versus threading in C/C++". In: *Concurrency - Practice and Experience* 12.12 (2000), pp. 1165–1176.

[82] K. Lakshmanan, S. Kato, and R. Rajkumar. "Scheduling Parallel Real-Time Tasks on Multi-core Processors". In: *IEEE Real-Time Systems Symposium*. 2010, pp. 259–268.

[83] S. Lee, S.-J. Min, and R. Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization". In: *ACM Sigplan Notices* 44.4 (2009), pp. 101–110.

[84] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, et al. "Towards fully autonomous driving: Systems and algorithms". In: *Intelligent Vehicles Symposium (IV)*. IEEE. 2011, pp. 163–168.

[85] J. Li, D. Hei, and L. Yan. "Correctness analysis based on testing and checking for openmp programs". In: *ChinaGrid Annual Conference*. IEEE. 2009, pp. 210–215.

[86] C. Liao, D. J. Quinlan, T. Panas, and B. R. De Supinski. "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries". In: *International Workshop on OpenMP*. Springer. 2010, pp. 15–28.

[87] Y. Lin. "Static nonconcurrency analysis of openmp programs". In: *International Workshop on OpenMP*. Springer, 2008, pp. 36–50.

[88] E. Lippe and N. van Oosterom. "Operation-based Merging". In: *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*. SDE 5. Tyson's Corner, Virginia, USA: ACM, 1992, pp. 78–87.

[89] T. Liu, Z. Ji, and Q. Wang. "Research on OpenMP algorithms on memory limited embedded multicore platform". In: *Journal of Computational Information Systems* 6.13 (2010), pp. 4453–4460.

[90] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia. "Servicess: An interoperable programming framework for the cloud". In: *Journal of Grid Computing* 12.1 (2014), pp. 67–91.

[91] M. Lundstrom. "Moore's law forever?" In: *Science* 299.5604 (2003), pp. 210–211.

[92] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford. "Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs". In: *International Conference on Field Programmable Logic and Applications*. IEEE. 2006, pp. 1–6.

[93] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang. "Symbolic analysis of concurrency errors in OpenMP programs". In: *International Conference on Parallel Processing*. IEEE. 2013, pp. 510–516.

[94] A. Marongiu, P. Burgio, and L. Benini. "Supporting OpenMP on a multi-cluster embedded MPSoC". In: *Microprocessors and Microsystems* 35.8 (2011), pp. 668–682.

[95] T. J. McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[96] A. Melani, M. A. Serrano, M. Bertogna, I. Cerutti, E. Quiñones, and G. Buttazzo. "A static scheduling approach to enable safety-critical OpenMP applications". In: *Asia and South Pacific Design Automation Conference*. IEEE. 2017, pp. 659–665.

[97] Meyer, Bertrand. *Object-oriented software construction*. Vol. 2. Prentice hall New York, 1988.

[98] S. Michell, B. Moore, and L. M. Pinho. "Tasklettes – a fine grained parallelism for Ada on multicores". In: *Ada-Europe International Conference on Reliable Software Technologies*. Springer. 2013, pp. 17–34.

[99] M. Mohaqeqi, J. Abdullah, N. Guan, and W. Yi. "Schedulability analysis of synchronous digraph real-time tasks". In: *28th Euromicro Conference on Real-Time Systems*. Ed. by L. O'Conne. Toulouse, France: IEEE, 2016, pp. 176–186.

[100] B. Moore. "Paraffin: a Parallelism API for Multiple Languages". In: *Ada User Journal* 37.2 (2016).

[101] B. J. Moore. "Parallelism generics for Ada 2005 and beyond". In: *Ada Letters*. Vol. 30. 3. ACM. 2010, pp. 41–52. URL: http://paraffin.sourceforge.net.

[102] J. F. Münchhalfen, T. Hilbrich, J. Protze, C. Terboven, and M. S. Müller. "Classification of common errors in OpenMP applications". In: *International Workshop on OpenMP*. Springer. 2014, pp. 58–72.

[103] M. Naik, C.-S. Park, K. Sen, and D. Gay. "Effective static deadlock detection". In: *International Conference on Software Engineering*. IEEE. 2009, pp. 386–396.

[104] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[105] R. H. Netzer and B. P. Miller. "What are race conditions?: Some issues and formalizations". In: *Programming Languages and Systems* 1.1 (1992), pp. 74–88.

[106] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. "O'Reilly Media, Inc.", 1996.

[107] D.-I. Oh, T. P. Baker, and S.-J. Moon. "The GNARL implementation of POSIX/Ada signal services". In: *International Conference on Reliable Software Technologies*. Springer. 1996, pp. 275–286.

[108] OpenMP ARB. *OpenMP Application Program Interface, version 2.5*. 2005. URL: {http://www.openmp.org/wp-content/uploads/spec25.pdf}.

[109] OpenMP ARB. *OpenMP Application Program Interface, version 3.0*. 2008. URL: {http://www.openmp.org/wp-content/uploads/spec30.pdf}.

[110] OpenMP ARB. *OpenMP Application Program Interface, version 3.1*. 2011. URL: {http://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf}.

[111] OpenMP ARB. *OpenMP Application Program Interface, version 4.0*. 2013. URL: {http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf}.

[112] OpenMP ARB. *OpenMP Application Program Interface: Examples, version 4.5*. 2015. URL: {http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf}.

[113] OpenMP ARB. *OpenMP Application Program Interface, version 4.5*. 2015. URL: {http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf}.

[114] *Oracle Solaris Studio 12.3 OpenMP User's Guide*. 2012. URL: {{https://docs.oracle.com/cd/E24457_01/pdf/E21996.pdf}}.

[115] N. Papakonstantinou, F. S. Zakkak, and P. Pratikakis. “Hierarchical Parallel Dynamic Dependence Analysis for Recursively Task-Parallel Programs”. In: *Parallel and Distributed Processing Symposium*. IEEE. 2016, pp. 933–942.

[116] *Paraffin*. 2017. URL: {http://paraffin.sourceforge.net}.

[117] P. Petersen and S. Shah. “OpenMP support in the Intel® thread checker”. In: *International Workshop on OpenMP Applications and Tools*. Springer. 2003, pp. 1–12.

[118] PGI®. *Fortran Reference Guide*. 2017. URL: {https://www.pgroup.com/resources/docs/17.9/pdf/pgi17fortref.pdf}.

[119] L. M. Pinho and S. Michell. “Session Summary: Parallel and Multicore Systems”. In: *Ada Lett.* 36.1 (2016), pp. 83–90.

[120] L. M. Pinho, B. Moore, S. Michell, and S. T. Taft. “Real-Time Fine-Grained Parallelism in Ada”. In: *ACM SIGAda Ada Letters* 35.1 (2015), pp. 46–58.

[121] L. M. Pinho, V. Nélis, P. M. Yomsi, E. Quiñones, M. Bertogna, P. Burgio, A. Marongiu, C. Scordino, P. Gai, M. Ramponi, et al. “P-SOCRATES: A parallel software framework for time-critical many-core systems”. In: *Microprocessors and Microsystems* 39.8 (2015), pp. 1190–1203.

[122] A. Podobas and S. Karlsson. “Towards Unifying OpenMP Under the Task-Parallel Paradigm”. In: *International Workshop on OpenMP*. 2016, pp. 116–129.

[123] Project Hi-Lite. *GNATprove*. http://www.open-do.org/projects/hi-lite/gnatprove. 2017.

[124] X. Qi and B. Xu. “An approach to slicing concurrent Ada programs based on program reachability graphs”. In: *International Journal of Computer Science and Network Security* 6.1 (2005), pp. 29–37.

[125] F. M. Quintao Pereira, R. E. Rodrigues, and V. H. Sperle Campos. “A fast and low-overhead technique to secure programs against integer overflows”. In: *International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2013, pp. 1–11.

[126] W. River. *VxWorks programmer’s guide*. 2003. URL: {http://www.windriver.com}.

[127] C. Rochange, A. Bonenfant, P. Sainrat, M. Gerdes, J. Wolf, T. Ungerer, Z. Petrov, and F. Mikulu. “WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core”. In: *OASIcs-OpenAccess Series in Informatics*. Vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[128] T. Rothwell and J. Youngman. *The GNU C Reference Manual*. 2016. URL: {https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf}.

[129] S. Royuela, A. Duran, C. Liao, and D. J. Quinlan. “Auto-scoping for OpenMP Tasks”. In: *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*. Rome, Italy: Springer-Verlag, 2012, 29–43.

[130] S. Royuela, A. Duran, and X. Martorell. “Compiler Analysis and its application to OmpSs”. PhD thesis. Universitat Politècnica de Catalunya · BarcelonaTech (UPC), 2012.

[131] S. Royuela, A. Duran, and X. Martorell. “Compiler automatic discovery of ompss task dependencies”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2012, pp. 234–248.

[132] S. Royuela, A. Duran, M. A. Serrano, E. Quiñones, and X. Martorell. “A Functional Safety OpenMP* for Critical Real-Time Embedded Systems”. In: *Scaling OpenMP for Exascale Performance and Portability – 13th International Workshop on OpenMP*. Ed. by B. R. de Supinski, O. S. L., C. Terboven, B. M. Chapman, and M. S. Müller. Stony Brook, NY, USA: Springer, 2017, pp. 231–245.

[133] S. Royuela, R. Ferrer, D. Caballero, and X. Martorell. “Compiler analysis for OpenMP tasks correctness”. In: *International Conference on Computing Frontiers*. ACM. 2015.

[134] S. Royuela, X. Martorell, E. Quiñones, and L. M. Pinho. “OpenMP tasking model for Ada: safety and correctness”. In: *Ada-Europe International Conference on Reliable Software Technologies*. Springer. 2017.

[135] S. Rybin, A. Strohmeier, and E. Zueff. “ASIS for GNAT: Goals, Problems and Implementation Strategy”. In: *ACM SIGAda Ada Letters* 16.2 (1996), pp. 39–49.

[136] V. Sarkar and J. Hennessy. “Compile-time partitioning and scheduling of parallel programs”. In: *ACM Sigplan Notices*. Vol. 21. 7. ACM. 1986, pp. 17–26.

[137] M. Schmidt, D. Fey, and M. Reichenbach. “Parallel Embedded Computing Architectures”. In: *Embedded Systems-High Performance Systems, Applications and Projects*. InTech, 2012.

[138] M. A. Serrano, A. Melani, M. Bertogna, and E. Quiñones. “Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions”. In: *Conference on Design, Automation & Test in Europe*. IEEE. 2016, pp. 1066–1071.

[139] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones. “Timing characterization of OpenMP4 tasking model”. In: *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press. 2015, pp. 157–166.

[140] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. “Performance gaps between OpenMP and OpenCL for multi-core CPUs”. In: *International Conference on Parallel Processing Workshops*. IEEE. 2012, pp. 116–125.

[141] E. Srl. “Erika enterprise”. In: (2017). URL: {erika.tuxfamily.org}.

[142] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. “A Scalable Approach to Thread-level Speculation”. In: *International Symposium on Computer Architecture*. Vancouver, Canada: ACM, 2000, pp. 1–12.

[143] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in Science & Engineering* 12.3 (2010), pp. 66–73.

[144] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault. “Openmp on the low-power ti keystone ii arm/dsp system-on-chip”. In: *International Workshop on OpenMP*. Springer. 2013, pp. 114–127.

[145] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi. “Scheduling and analysis of real-time openmp task systems with tied tasks”. In: *IEEE Real-Time Systems Symposium*. 2017.

[146] M. Süß and C. Leopold. “Common mistakes in OpenMP and how to avoid them”. In: *International Workshop on OpenMP*. Springer, 2008, pp. 312–323.

[147] S. T. Taft, B. Moore, L. M. Pinho, and S. Michell. “Safe parallel programming in Ada with language extensions”. In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 87–96.

[148] TERAFLUX Consortium. *TERAFLUX: Exploiting Dataflow Parallelism in Teradevice Computing*. 2014. URL: {http://www.teraflux.eu/}.

[149] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé. “Support for OpenMP tasks in Nanos v4”. In: *Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp. 2007, pp. 256–259.

[150] The P-SOCRATES Consortium. *The P-SOCRATES Project*. 2014. URL: http://www.p-socrates.eu/.

[151] D. Theodoropoulos, D. Pnevmatikatos, C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, X. Martorell, N. Navarro, C. Segura, C. Fernandez, D. Oro, J. Rodriguez Saeta, P. Gai, A. Rizzo, and R. Giorgi. “The AXIOM project (agile, extensible, fast i/o module)”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE. 2015, pp. 262–269.

[152] G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, and D. S. Nikolopoulos. “BDDT: block-level dynamic dependence analysis for deterministic task-based parallelism”. In: *ACM SIGPLAN Notices*. Vol. 47. 8. New York, NY, USA: ACM, 2012, pp. 301–302.

[153] U. P. d. C. ·. B. (UPC). *Parallelism*. 2014. URL: {http://www.fib.upc.edu/en/estudiar-enginyeria-informatica/assignatures/PAR.html}.

[154] A. L. Varbanescu, P. Hijma, R. Van Nieuwpoort, and H. Bal. “Towards an effective unified programming model for many-cores”. In: *Parallel and Distributed Processing Workshops and Phd Forum*. IEEE. 2011, pp. 681–692.

[155] R. Vargas, E. Quiñones, and A. Marongiu. “OpenMP and timing predictability: a possible union?” In: *Conference on Design, Automation & Test in Europe*. EDA Consortium. 2015, 617–620.

[156] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quiñones. "A lightweight OpenMP4 run-time for embedded systems". In: *Asia and South Pacific Design Automation Conference*. IEEE. 2016, pp. 43–49.

[157] T. Vijaykumar and G. S. Sohi. "Task selection for a multiscalar processor". In: *International Symposium on Microarchitecture*. IEEE Computer Society Press. 1998, pp. 81–92.

[158] B. Wang, H. Gao, and J. Cheng. "Definition-Use Net and System Dependence Net generators for Ada 2012 programs and their applications". In: *Ada User Journal* 38.1 (2017), pp. 37–55.

[159] C. Wang, S. Chandrasekaran, B. Chapman, and J. Holt. "libEOMP: a portable OpenMP runtime library based on MCA APIs for embedded systems". In: *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM. 2013, pp. 83–92.

[160] C.-K. Wang and P.-S. Chen. "Automatic scoping of task clauses for the OpenMP tasking model". In: *The Journal of Supercomputing* 71.3 (2015), pp. 808–823.

[161] Y. Wang, N. Guan, J. Sun, M. Lv, Q. He, T. He, and W. Yi. "Benchmarking OpenMP programs for real-time scheduling". In: *Embedded and Real-Time Computing Systems and Applications*. IEEE. 2017, pp. 1–10.

[162] B. A. Wichmann. "Guide for the Use of the Ada Programming Language in High Integrity Systems". In: *ACM SIGAda Ada Letters* XVIII.4 (1998), pp. 47–94.

[163] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Pauaut, P. Puschner, J. Staschulat, and P. Stenström. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems* 7.3 (2008), p. 36.

[164] M. Wong, M. Klemm, A. Duran, T. Mattson, G. Haab, B. R. de Supinski, and A. Churbanov. "Towards an error model for OpenMP". In: *International Workshop on OpenMP*. Springer. 2010, pp. 70–82.

[165] F. S. Zakkak, D. Chasapis, P. Pratikakis, A. Bilas, and D. S. Nikolopoulos. "Inference and declaration of independence in task-parallel programs". In: *International Workshop on Advanced Parallel Processing Technologies*. Springer. 2013, pp. 1–16.

Figures

2.1 Time-line of the OpenMP releases. 6
2.2 Producer-consumer pattern implemented using OpenMP flush constructs. 8
2.3 Diagram of Ada task states. 11
2.4 Mercurium compilation diagram. 16
2.5 Mercurium simplified AST for Fibonacci computation in Listing 2.7. 17
2.6 Diagram of an SMP system. 21
2.7 Diagram of a NUMA system. 21
2.8 High-level view of the Intel Xeon E5-2670. 22
2.9 High-level view of the Intel Xeon Platinum 8160. 22
2.10 High-level view of the MPPA processor. 23
2.11 High-level view of an MPPA compute cluster. 24
2.12 High-level view of an MPPA VLIW core. 24

3.1 Examples of functions $\mathcal{PCFG}_{simple}$ and $\mathcal{PCFG}_{struct}$, which build the PCFG. . . 27
3.2 Process that determines if two tasks synchronize. 28
3.3 PCFG for code in Listing 3.1. 32
3.4 Equations that determine the liveness attributes of a PCFG node. 35
3.5 Equations that determine the reaching definitions of a PCFG node. 35
3.6 Example illustrating the impact of OpenMP tasks regarding reaching definitions: (a) code snippet (b) simplified PCFG. 36

4.1 Scenario illustrating an automatic storage variable that may be accessed after its lifetime ends. 42
4.2 Scenario illustrating a race condition due to a wrong synchronization of a task. . . 43
4.3 Scenario illustrating a useless definition of dependences between non-sibling tasks. 45
4.4 Scenario illustrating different incoherences in the data-sharing attributes of a task. 47
4.5 Scenario illustrating the wrong specification of task dependences. 48
4.6 Occurrences of different correctness mistakes. 50

5.1 fTDG of the OpenMP program in Listing 5.1. 59
5.2 TDG of the OpenMP program in Listing 5.1. 60
5.3 Hash table that stores the TDG depicted in Figure 5.2, corresponding to the OpenMP program in Listing 5.1. 62

5.4 Performance speed-up and memory usage (in KB) of Cholesky and r3DPP applications running with *lightweight omp4, omp4* and *omp 3.1*, and varying the number of tasks. 63

6.1 Example of propagation of the usage and global directives over a call graph. . . . 79
6.2 Concurrency available with the Ada parallel model (left) and OpenMP tasks (right). 84
6.3 Mapping nested parallelism between Ada and OpenMP. 88
6.4 Performance speedup of the Ada parallel programming model implemented with OpenMP, Ada tasks and Paraffin. 90
6.5 TDG of Cholesky kernel implemented using tasks and taskwaits. 92
6.6 TDG of Cholesky kernel implemented using tasks with dependences. 92
6.7 Speedup of Ada/OpenMP (structured parallelism) and OpenMP with dependences (unstructured parallelism). 92
6.8 Execution time of OpenMP running with Ada and C. 93
6.9 Execution trace of the OpenMP and Ada tasks mixed benchmark. 93
6.10 HRT-HOOD representation of the *Ravenscar* application defined in Section 7 of the “Ada Ravenscar Profile Guide”. 96
6.11 Control flow graph of the *Ravenscar* application defined in Section 7 of the “Ada Ravenscar Profile Guide”. 99
6.12 Control flow graph of the OpenMP code introduced in the *Small_Whetstone* procedure from the *Ravenscar* application defined in Section 7 of the “Ada Ravenscar Profile Guide”. 100
6.13 Schema of the issues addressed in the context of applying OpenMP in the safety-critical domain by means of Ada. 103

A.1 Diagram of Ada task states and transitions 133

Tables

4.1 Oracle Solaris Studio and Mercurium messages for different correctness situations. 52

5.1 Minimum memory (in Bytes) used to store an OpenMP task in different runtimes. 56
5.2 Memory usage of the sparse matrix (in KB), varying the number of tasks instantiated. 64

6.1 Solutions for race conditions in an Ada/OpenMP application. 98

Listings

2.1 Addition of two arrays using the OpenMP thread-centric model. 7
2.2 Addition of two arrays using the OpenMP task-centric model. 7
2.3 OmpSs multidependences syntax example. 10
2.4 Stock data-structure implemented using synchronization-free Ada protected objects. 13
2.5 Stock data-structure implemented using Ada protected objects with synchronization. 13
2.6 Stock data-structure implemented using the Ada rendezvous mechanism. 13
2.7 Recursive Fibonacci computation using OpenMP tasks. 17
2.8 Example of rules of the grammar generating the Mercurium's IR. 18
2.9 Example of a configuration file of Mercurium. 19
3.1 Matrix multiplication using OpenMP tasks (Example *task_dep.5.c* from the specification examples [112]) 31
3.2 OpenMP example illustrating the propagation of usage information to outer nodes. 34
5.1 OpenMP tasks example traversing a matrix with a wavefront strategy. 57
6.1 Example of OpenMP function using several constructs. 77
6.2 Function declaration of method in Listing 6.1 using the proposed extensions for safety-critical systems. 77
6.3 Factorial computation parallelized with OpenMP. 77
6.4 Function declaration for method in Listing 6.3 using the extensions for safety-critical OpenMP. 77
6.5 Ada syntax for parallel blocks. 82
6.6 Ada syntax for a parallel loop. 82
6.7 Ada syntax for a not chuncked parallel. 82
6.8 Parallel reduction with proposed Ada extensions. 83
6.9 OpenMP syntax for parallel blocks. 84
6.10 OpenMP syntax for a parallel loop. 84
6.11 Example of Ada periodic task using a `delay` statement. 94
6.12 OpenMP code inserted in the *Production_Workload* package of the *Ravenscar* application. 97
B.1 Incomplete parallel computation of the Fibonacci sequence using OpenMP. 135
B.2 Incomplete parallel computation of several Fibonacci sequences, one per each element of a linked list, using OpenMP. 136
B.3 Incomplete parallel computation of a dot product, using OpenMP. 137
B.4 Incomplete parallel matrix multiplication, using OpenMP (main and kernel methods). 138

B.5 Incomplete parallel matrix multiplication, using OpenMP (support methods). . . . 139
B.6 Incomplete parallel computation of number pi approximating the area under the curve $f(x) = 4/(1 + x * x)$ between 0 and 1, using OpenMP. 140
B.7 Incomplete parallel computation of a Sudoku problem using OpenMP (kernel). . . 141
B.8 Incomplete parallel computation of a Sudoku problem using OpenMP (main function). 142
B.9 Cholesky kernel implemented with C. 143
B.10 Cholesky kernel implemented with C and OpenMP tasks, and synchronizing tasks with taskwaits. 143
B.11 Cholesky kernel implemented with C and OpenMP tasks, and synchronizing tasks with dependences. 144
B.12 Cholesky kernel implemented with Ada. 145
B.13 Cholesky kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with taskwaits. 145
B.14 Cholesky kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with dependences. 146
B.15 Cholesky kernel implemented with Ada tasks: synchronization mechanisms. . . . 147
B.16 Cholesky kernel implemented with Ada tasks: secondary kernels. 148
B.17 Cholesky kernel implemented with Ada tasks: main kernel. 149
B.18 Cholesky kernel implemented with Paraffin. 150
B.19 LU kernel implemented with C. 151
B.20 LU kernel implemented with C and OpenMP tasks, and synchronizing tasks with taskwaits. 151
B.21 LU kernel implemented with C and OpenMP tasks, and synchronizing tasks with dependences. 152
B.22 LU kernel implemented with Ada. 152
B.23 LU kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with taskwaits. 153
B.24 LU kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with dependences. 154
B.25 LU kernel implemented with Ada tasks: synchronization mechanisms. 155
B.26 LU kernel implemented with Ada tasks: secondary kernels. 156
B.27 LU kernel implemented with Ada tasks: more secondary kernels. 157
B.28 LU kernel implemented with Ada tasks: main kernel. 158
B.29 LU kernel implemented with Ada and Paraffin: synchronization mechanisms. . . 159
B.30 Matrix kernel implemented with C. 160
B.31 Matrix kernel implemented with C and OpenMP tasks, and synchronizing tasks with taskwaits. 160
B.32 Matrix kernel implemented with Ada. 161
B.33 Matrix kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with taskwaits. 161

B.34 Matrix kernel implemented with Ada tasks. 162
B.35 Matrix kernel implemented with Paraffin. 163
B.36 Synthetic kernel with interaction between Ada tasks and OpenMP tasks, using Extrae instrumentation tool: Ada tasks. 164
B.37 Synthetic kernel with interaction between Ada tasks and OpenMP tasks, using Extrae instrumentation tool: OpenMP tasks. 165
B.38 Synthetic kernel with interaction between Ada tasks and OpenMP tasks, using Extrae instrumentation tool: global objects and main function. 166

List of Algorithms

1	High-level algorithm for synchronizing tasks within a PCFG.	31
2	High-level algorithm to detect accesses to variables in OpenMP tasks whose lifetime may have ended.	42
3	High-level algorithm to detect race conditions in OpenMP tasks.	44
4	High-level algorithm to detect dependences among non-sibling OpenMP tasks. . .	46
5	High-level algorithm to detect incoherences in the data-sharing attributes of OpenMP tasks.	47
6	High-level algorithm to detect incoherences in the dependence clauses of OpenMP tasks.	49
7	Rules to determine the clauses of the `usage` directive to be added to the contract of a safety-critical function.	76
8	High-level algorithm to compute the contracts of a safety-critical OpenMP library.	78
9	High-level algorithm to detect race conditions in Ada/OpenMP programs.	101

Diagrams

This appendix contains extended diagrams.

A.1 Ada task states and transitions

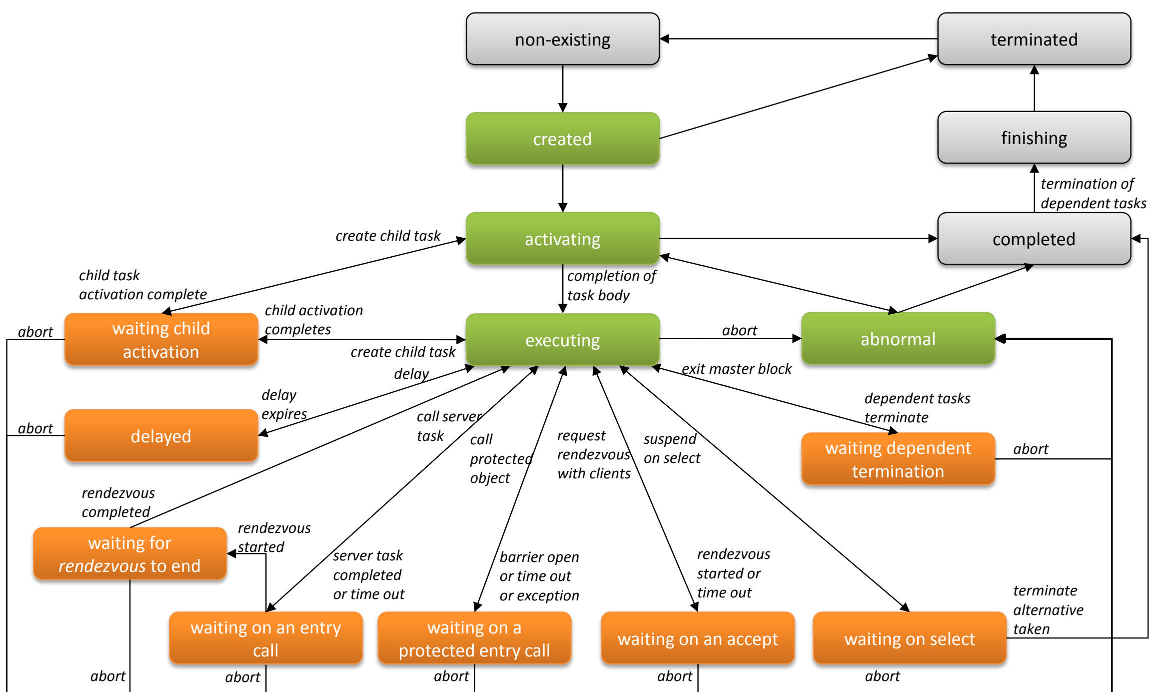


Figure A.1: Diagram of Ada task states and transitions

B

Benchmark Source Codes

This appendix contains the source code of the benchmarks used in this thesis.

B.1 Benchmarks for correctness checking in OpenMP

B.1.1 Fibonacci

```
1  long long fib (int n)
2  {
3      long long x, y;
4      if (n < 2) return n;
5
6      #pragma omp task    // shared(x) firstprivate(n)
7          x = fib(n - 1);
8
9      #pragma omp task    // shared(y) firstprivate(n)
10         y = fib(n - 2);
11
12     // pragma omp taskwait
13
14     return x + y;
15 }
16
17 int main(int argc, char** argv)
18 {
19     const char Usage[] = "Usage: fib <num> (try 20)\n";
20     if (argc < 2) {
21         fprintf(stderr, Usage);
22         exit(1);
23     }
24
25     int num = atoi(argv[1]);
26     long long res;
27     #pragma omp parallel
28     #pragma omp single
29         res = fib(num);
30
31     printf("The Fibonacci number of %s is %lld\n", argv[1], res);
32
33     return 0;
34 }
```

Listing B.1: Incomplete parallel computation of the Fibonacci sequence using OpenMP.

```
1  #define N 5
2  #define FS 1
3  struct node { int data; int fibdata; struct node* next; };
4
5  int fib(int n) {
6      if (n < 2) return (n);
7      else return fib(n - 1) + fib(n - 2);
8  }
9
10 void processwork(struct node* p) {
11     p->fibdata = fib(p->data);
12 }
13
14 struct node* init_list(struct node* p) {
15     struct node* head = malloc(sizeof(struct node));
16     p = head;
17     p->data = FS;
18     p->fibdata = 0;
19     for (int i = 0; i < N; i++) {
20         struct node* temp  = malloc(sizeof(struct node));
21         p->next = temp;
22         p = temp;
23         p->data = FS + i + 1;
24         p->fibdata = i+1;
25     }
26     p->next = NULL;
27     return head;
28 }
29
30 int main() {
31     struct node *p, *temp, *head;
32     p = init_list(p);
33     head = p;
34
35     #pragma omp parallel
36     #pragma omp single
37     {
38         p = head;
39         while (p) {
40             #pragma omp task // firstprivate(p)
41                 processwork(p);
42             p = p->next;
43         }
44     }
45
46     p = head;
47     while (p != NULL) {
48         temp = p->next;
49         free(p);
50         p = temp;
51     }
52     free(p);
53     return 0;
54 }
```

Listing B.2: Incomplete parallel computation of several Fibonacci sequences, one per each element of a linked list, using OpenMP.

B.1.2 Dot product

```
long N, CHUNK_SIZE;
static void initialize(long length, double data[length])
{
    for (long i = 0; i < length; i++)
        data[i] = ((double)rand() / (double)RAND_MAX);
}

double dot_product (long N, long CHUNK_SIZE, double A[N], double B[N])
{
    long N_CHUNKS = N / CHUNK_SIZE;
    if (N_CHUNKS * CHUNK_SIZE < N) N_CHUNKS++;
    double *C = malloc(N_CHUNKS * sizeof(double));
    double acc = 0.0;
    int j = 0;
    long actual_size = (N - CHUNK_SIZE >= CHUNK_SIZE) ? CHUNK_SIZE
                                                      : N - CHUNK_SIZE;
    for (long i = 0; i < N; i += CHUNK_SIZE) {
        #pragma omp task firstprivate(j, i, actual_size)
        // depend(in:A[i:i+actual_size-1], B[i:i+actual_size-1])
        // depend(inout:C[j])
        {
            C[j]=0;
            for (long ii=0; ii<actual_size; ii++)
                C[j] += A[i+ii] * B[i+ii];
        }
        #pragma omp task firstprivate(j) // shared(acc) inout(acc) in(C[j])
            acc += C[j];
        j++;
    }
    // #pragma omp taskwait
    return(acc);
}

int main(int argc, char **argv)
{
    N = atol(argv[1]) * 1024L;
    CHUNK_SIZE = atol(argv[2]) * 1024L;

    double *A = malloc(N*sizeof(double));
    double *B = malloc(N*sizeof(double));
    initialize(N, A);
    initialize(N, B);

    double result;
    #pragma omp parallel
    #pragma omp single
    result = dot_product (N, CHUNK_SIZE, A, B);

    printf ("Result of Dot product: %le\n", result);
    return 1;
}
```

Listing B.3: Incomplete parallel computation of a dot product, using OpenMP.

B.1.3 Matrix multiplication

```
1 void matmul(double *A, double *B, double *C, unsigned long NB)
2 {
3     unsigned I;
4     double tmp;
5     for (unsigned i = 0; i < NB; i++)
6     {
7         I = i * NB;
8         for (unsigned j = 0; j < NB; j++)
9         {
10            tmp = C[I+j];
11            for (unsigned k = 0; k < NB; k++)
12                tmp += A[I+k] * B[k*NB+j];
13            C[I+j] = tmp;
14        }
15    }
16 }
17
18 void compute(unsigned long NB, unsigned long DIM,
19              double *A[DIM][DIM], double *B[DIM][DIM], double *C[DIM][DIM])
20 {
21     for (unsigned i = 0; i < DIM; i++)
22         for (unsigned j = 0; j < DIM; j++)
23             for (unsigned k = 0; k < DIM; k++)
24                 #pragma omp task
25                 // in(A[i][k], B[k][j]) inout(C[i][j])
26                 matmul((double*)A[i][k], (double*)B[k][j], (double*)C[i][j], NB);
27
28     // #pragma omp taskwait
29 }
30
31 int main(int argc, char *argv[])
32 {
33     if (argc != 3) {
34       printf("usage: %s DIM NB\n", argv[0]);
35       exit(0);
36     }
37
38     unsigned long DIM = atoi(argv[1]);
39     unsigned long NB = atoi(argv[2]);
40     unsigned long N = NB * DIM;
41     double **A, **B, **C;
42     init(&A, &B, &C, N, DIM, NB);
43
44     compute(NB, DIM, (void*)A, (void*)B, (void*)C);
45
46     return 0;
47 }
```

Listing B.4: Incomplete parallel matrix multiplication, using OpenMP (main and kernel methods).

```
1  void convert_to_blocks(unsigned long NB, unsigned long DIM,
2                         unsigned long N,
3                         double *Alin, double *A[DIM][DIM])
4  {
5      for (unsigned i = 0; i < N; i++)
6          for (unsigned j = 0; j < N; j++)
7              A[i/NB][j/NB][(i%NB)*NB + j%NB] = Alin[j*N + i];
8  }
9
10 void fill_random(double *Alin, int NN)
11 {
12     for (int i = 0; i < NN; i++)
13         Alin[i] = ((double)rand()) / ((double)RAND_MAX);
14 }
15
16 void init (double ***A, double ***B, double ***C,
17            unsigned long N, unsigned long DIM, unsigned long NB)
18 {
19     double *Alin = (double *) malloc(N * N * sizeof(double));
20     double *Blin = (double *) malloc(N * N * sizeof(double));
21     double *Clin = (double *) malloc(N * N * sizeof(double));
22
23     srand(0);
24     fill_random(Alin, N * N);
25     fill_random(Blin, N * N);
26     for (int i = 0; i < N * N; i++)
27       Clin[i] = 0.0;
28
29     *A = (double **) malloc(DIM * DIM * sizeof(double *));
30     *B = (double **) malloc(DIM * DIM * sizeof(double *));
31     *C = (double **) malloc(DIM * DIM * sizeof(double *));
32
33     for (int i = 0; i < DIM*DIM; i++) {
34         (*A)[i] = (double *) malloc(NB * NB * sizeof(double));
35         (*B)[i] = (double *) malloc(NB * NB * sizeof(double));
36         (*C)[i] = (double *) malloc(NB * NB * sizeof(double));
37     }
38     convert_to_blocks(NB, DIM, N, Alin, (double*(*)[DIM])(*A));
39     convert_to_blocks(NB, DIM, N, Blin, (double*(*)[DIM])(*B));
40     convert_to_blocks(NB, DIM, N, Clin, (double*(*)[DIM])(*C));
41
42     free(Alin);
43     free(Blin);
44     free(Clin);
45 }
```

Listing B.5: Incomplete parallel matrix multiplication, using OpenMP (support methods).

B.1.4 Pi

```
int main(int argc, char *argv[])
{
    double x, sum=0.0, pi=0.0;
    int i;

    const char Usage[] = "Usage: pi <num_steps> (try 1000000000)\n";
    if (argc < 2) {
        fprintf(stderr, Usage);
        exit(1);
    }

    int num_steps = atoi(argv[1]);
    double step = 1.0/(double) num_steps;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        // private(i,x) shared(sum)
        for (i=0; i < num_steps/2; i++) {
            x = (i+0.5)*step;
            // #pragma omp atomic
            sum += 4.0/(1.0+x*x);
        }

        #pragma omp task
        // private(i,x) shared(sum)
        for (i=num_steps/2; i < num_steps; i++) {
            x = (i+0.5)*step;
            // #pragma omp atomic
            sum += 4.0/(1.0+x*x);
        }

        // #pragma omp taskwait

        #pragma omp task
        pi = step * sum;
    }

    printf("Value of pi = %12.10f\n", pi);

    return EXIT_SUCCESS;
}
```

Listing B.6: Incomplete parallel computation of number pi approximating the area under the curve $f(x) = 4/(1 + x * x)$ between 0 and 1, using OpenMP.

B.1.5 Sudoku solver

```
1  unsigned long num_solutions = 0;
2  int* first_solution = NULL;
3
4  int solve(int size, int* g, int loc)
5  {
6      int i, num_guesses, solved=0;
7      int len = size*size*size*size;
8      int allGuesses[size*size];
9
10     if (loc == len) {
11         // #pragma omp atomic
12         num_solutions++;
13         // #pragma omp critical
14         if (!first_solution)
15             first_solution = new_grid(size, g);
16         return 1;
17     }
18
19     if (g[loc] != 0) {
20         solved = solve(size, g, loc+1);
21         return solved;
22     }
23
24     all_guesses(size, loc, g, allGuesses, &num_guesses);
25     for (i = 0; i < num_guesses; i++) {
26         if (loc < 10) {
27             #pragma omp task
28             // shared(allGuesses, solved)
29             {
30                 int *help = new_grid(size, g);
31                 help[loc] = allGuesses[i];
32                 if (solve(size, help, loc+1))
33                     // #pragma omp critical
34                     solved = 1;
35                 free(help);
36             }
37         }
38         else {
39           g[loc] = allGuesses[i];
40           if (solve(size, g, loc+1))
41               // #pragma omp critical
42               solved = 1;
43           g[loc] = 0;
44         }
45     }
46     // #pragma omp taskwait
47
48     return solved;
49 }
```

Listing B.7: Incomplete parallel computation of a Sudoku problem using OpenMP (kernel).

```
1 int main(int argc, char **argv) {
2     int solved;
3     int size;
4 
5     if (argc != 2) {
6         fprintf(stderr, "Usage: %s <puzzle_filename> \n", argv[0]);
7         return(0);
8     }
9 
10    FILE* fd = fopen(argv[1], "r");
11    if (fd == NULL) {
12        printf("Error: Failed to open file with initial puzzle\n");
13        return(0);
14    }
15
16    solved = fscanf(fd, "%d", &size);
17    int* g = new_grid(size, NULL);
18
19    read_puzzle(size, g, fd);
20    printf("\nInitial puzzle (size %d):\n", size);
21    write_puzzle(size, g);
22
23    #pragma omp parallel
24    #pragma omp single
25        solved = solve(size, g, 0);
26
27    if (solved == 1)
28        printf("\nFound %lu solutions, first one being:\n", num_solutions);
29    else
30        printf("\nFailed to find a solution\n");
31
32    return(!solved);
33 }
```

Listing B.8: Incomplete parallel computation of a Sudoku problem using OpenMP (main function).

B.2 Benchmarks for the OpenMP integration into Ada

B.2.1 Cholesky decomposition

B.2.1.1 C

```
1 void cholesky_c_sequential(double M[NB][NB][BS*BS]) {
2     for (int k = 0; k < NB; k++) {
3         omp_potrf(M[k][k], BS, BS);
4         for (int i = k + 1; i < NB; i++)
5             omp_trsm(M[k][k], M[k][i], BS, BS);
6         for (int i = k + 1; i < NB; i++) {
7             for (int j = k + 1; j < i; j++)
8                 omp_gemm(M[k][i], M[k][j], M[j][i], BS, BS);
9             omp_syrk(M[k][i], M[i][i], BS, BS);
10        }
11    }
12 }
```

Listing B.9: Cholesky kernel implemented with C.

B.2.1.2 C + OpenMP

```
1 void cholesky_c_task_taskwaits(struct parallel_data *args) {
2     double (**M)[NB][BS*BS] = (*args).M;
3     if (omp_get_thread_num() == 0) {
4         for (int k = 0; k < NB; k++) {
5             omp_potrf((*M)[k][k], BS, BS);
6             for (int i = k + 1; i < NB; i++) {
7                 struct omp_trsm_data targs;
8                 targs.M = M; targs.k = k; targs.i = i;
9                 GOMP_task((void (*)(void *))omp_trsm_task,
10                          &targs, (void (*)(void *, void *))0, 24, 8, 1,
11                          GOMP_TASK_UNTIED, 0, 0);
12            }
13            GOMP_taskwait();
14
15            for (int i = k + 1; i < NB; i++) {
16                for (int j = k + 1; j < i; j++) {
17                    struct omp_gemm_data targs;
18                    targs.M = M; targs.k = k; targs.i = i; targs.j = j;
19                    GOMP_task((void (*)(void *))omp_gemm_task,
20                              &targs, (void (*)(void *, void *))0, 32, 8, 1,
21                              GOMP_TASK_UNTIED, 0, 0);
22                }
23                GOMP_taskwait();
24                omp_syrk((*M)[k][i], (*M)[i][i], BS, BS);
25            }
26        }
27    }
28 }
```

Listing B.10: Cholesky kernel implemented with C and OpenMP tasks, and synchronizing tasks with taskwaits.

```
1 void cholesky_c_task_dependences(struct parallel_data *args) {
2     double (**M)[NB][BS*BS] = (*args).M;
3     if (omp_get_thread_num() == 0) {
4         for (int k = 0; k < NB; k++) {
5             struct omp_potrf_data targs;
6             targs.M = M; targs.k = k;
7             void *deps[3L] = {[0] = (void *)1U, [1] = (void *)1U,
8                               [2] = (*M)[k][k]};
9             GOMP_task((void (*)(void *))omp_potrf_task,
10                      &targs, (void (*)(void *, void *))0, 32, 8, 1,
11                      UNTIED_DEPEND, deps, 0);
12            for (int i = k + 1; i < NB; i++) {
13                struct omp_trsm_data targs;
14                targs.M = M; targs.k = k; targs.i = i;
15                void *deps[4L] = {[0] = (void *)2U, [1] = (void *)1U,
16                                  [2] = (*M)[k][k], [3] = (*M)[k][i]};
17                GOMP_task((void (*)(void *))omp_trsm_task,
18                          &targs, (void (*)(void *, void *))0, 32, 8, 1,
19                          UNTIED_DEPEND, deps, 0);
20            }
21            for (int i = k + 1; i < NB; i++) {
22                for (int j = k + 1; j < i; j++) {
23                    struct omp_gemm_data targs;
24                    targs.M = M; targs.k = k; targs.i = i; targs.j = j;
25                    void *deps[5L] = {[0] = (void *)3U, [1] = (void *)1U,
26                                      [2] = (*M)[k][j], [3] = (*M)[k][j],
27                                      [4] = (*M)[j][i]};
28                    GOMP_task((void (*)(void *))omp_gemm_task,
29                              &targs, (void (*)(void *, void *))0, 40, 8, 1,
30                              UNTIED_DEPEND, deps, 0);
31                }
32                struct omp_syrk_data targs;
33                targs.M = M; targs.k = k; targs.i = i;
34                void *deps[4L] = {[0] = (void *)2U, [1] = (void *)1U,
35                                  [2] = (*M)[k][i], [3] = (*M)[i][i]};
36                GOMP_task((void (*)(void *))omp_syrk_task,
37                          &targs, (void (*)(void *, void *))0, 32, 8, 1,
38                          UNTIED_DEPEND, deps, 0);
39            }
40        }
41    }
42 }
```

Listing B.11: Cholesky kernel implemented with C and OpenMP tasks, and synchronizing tasks with dependences.

B.2.1.3 Ada

```
1 procedure cholesky_ada_sequential(M : in out Matrix_Type) is
2 begin
3     for k in 0 .. NB-1 loop
4         Omp_potrf(M(k, k), BS, BS);
5         for i in k + 1 .. NB-1 loop
6             Omp_trsm(M(k, k), M(k, i), BS, BS);
7         end loop;
8         for i in k + 1 .. NB-1 loop
9             for j in k + 1 .. i-1 loop
10                Omp_gemm(M(k, i), M(k, j), M(j, i), BS, BS);
11            end loop;
12            Omp_syrk(M(k, i), M(i, i), BS, BS);
13        end loop;
14    end loop;
15 end cholesky_ada_seq;
```

Listing B.12: Cholesky kernel implemented with Ada.

B.2.1.4 Ada + OpenMP

```
1 procedure cholesky_ada_taskwaits(args: System.Address) is
2   ... -- Function declarations
3   args_Acc: Parallel_Data_Type_Access := Convert_to_Parallel(args);
4   M_Acc: Matrix_Access := Convert_to_Matrix(args_Acc.M_Addr);
5   deps: OpenMP.Void_Ptr_Ptr := null;
6 begin
7   if (OpenMP.Ada_OMP_Get_Thread_Num = 0)
8   then
9     for k in 0 .. NB-1 loop
10       Omp_potrf(M_Acc.all(k, k), BS, BS);
11       for i in k + 1 .. NB-1 loop
12         ... -- Declarations
13         begin
14           targs.M_Addr := args_Acc.M_Addr; targs.k := k; targs.i := i;
15           OpenMP.Ada_GOMP_Task(omp_trsm_task'Unrestricted_Access, targs.all'Address,
16              null, 16, 8, TRUE, GOMP_TASK_UNTIED, deps, 0);
17         end;
18       end loop;
19       OpenMP.Ada_GOMP_Taskwait;
20       for i in k + 1 .. NB-1 loop
21         for j in k + 1 .. i-1 loop
22            ... -- Declarations
23            begin
24              targs.M_Addr := args_Acc.M_Addr; targs.k := k; targs.i := i; targs.j := j;
25              OpenMP.Ada_GOMP_Task(omp_gemm_task'Unrestricted_Access,
26                 targs.all'Address, null, 20, 8, TRUE, GOMP_TASK_UNTIED, deps, 0);
27            end;
28         end loop;
29         OpenMP.Ada_GOMP_Taskwait;
30         Omp_syrk(M_Acc.all(k, i), M_Acc.all(i, i), BS, BS);
31       end loop;
32     end loop;
33   end if;
34 end cholesky_ada_task_taskwaits;
```

Listing B.13: Cholesky kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with taskwaits.

```
1 procedure cholesky_ada_task_dependences(args: System.Address) is
2    ... -- Declarations
3 begin
4    if (OpenMP.Ada_OMP_Get_Thread_Num = 0) then
5       for k in 0 .. NB-1 loop
6          declare
7             targs: Omp_potrf_Access_Type := new Omp_potrf_Type;
8             deps_arr: aliased array(1..3) of aliased OpenMP.Void_Ptr :=
9                (PTR_1U, PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(k,k)));
10         begin
11            targs.M_Addr := args_Acc.M_Addr; targs.k := k;
12            OpenMP.Ada_GOMP_Task(omp_potrf_task'Unrestricted_Access, targs.all'Address,
13               null, 12, 8, TRUE, UNTIED_DEPEND, deps_arr(1)'Unrestricted_Access, 0);
14         end;
15         for i in k + 1 .. NB-1 loop
16            declare
17               targs: Omp_trsm_Access_Type := new Omp_trsm_Type;
18               deps_arr: aliased array(1..4) of aliased OpenMP.Void_Ptr :=
19                  (PTR_2U, PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(k,k))
20                  Submatrix_to_Void_Ptr(M_Acc.all(k,i)));
21            begin
22               targs.M_Addr := args_Acc.M_Addr; targs.k := k; targs.i := i;
23               OpenMP.Ada_GOMP_Task(omp_trsm_task'Unrestricted_Access, targs.all'Address,
24                  null, 16, 8, TRUE, UNTIED_DEPEND, deps_arr(1)'Unrestricted_Access, 0);
25            end;
26         end loop;
27         for i in k + 1 .. NB-1 loop
28            for j in k + 1 .. i-1 loop
29               declare
30                  targs: Omp_gemm_Access_Type := new Omp_gemm_Type;
31                  deps_arr: aliased array(1..5) of aliased OpenMP.Void_Ptr :=
32                     (PTR_3U, PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(k,i)),
33                     Submatrix_to_Void_Ptr(M_Acc.all(k,j)),
34                     Submatrix_to_Void_Ptr(M_Acc.all(j,i)));
35               begin
36                  targs.M_Addr := args_Acc.M_Addr;
37                  targs.k := k; targs.i := i; targs.j := j;
38                  OpenMP.Ada_GOMP_Task(omp_gemm_task'Unrestricted_Access,
39                     targs.all'Address, null, 20, 8, TRUE,
40                     UNTIED_DEPEND, deps_arr(1)'Unrestricted_Access, 0);
41               end;
42            end loop;
43            declare
44               targs: Omp_syrk_Access_Type := new Omp_syrk_Type;
45               deps_arr: aliased array(1..4) of aliased OpenMP.Void_Ptr :=
46                   (PTR_2U, PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(k,i)),
47                   Submatrix_to_Void_Ptr(M_Acc.all(i,i)));
48            begin
49               targs.M_Addr := args_Acc.M_Addr; targs.k := k; targs.i := i;
50               OpenMP.Ada_GOMP_Task(omp_syrk_task'Unrestricted_Access, targs.all'Address,
51                   null, 16, 8, TRUE, UNTIED_DEPEND, deps_arr(1)'Unrestricted_Access, 0);
52            end;
53         end loop;
54      end loop;
55   end if;
56 end cholesky_ada_task_dependences;
```

Listing B.14: Cholesky kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with dependences.

B.2.1.5 Ada tasks

```
1 procedure cholesky_ada_tasks(M : in out Matrix_Type) is
2     ... -- My_Barrier declaration
3     protected body My_Barrier is
4         entry Wait when Finished_Tasks = Num_Tasks is
5         begin
6             Finished_Tasks := 0;
7         end;
8         procedure Finished is
9         begin
10            Finished_Tasks := Finished_Tasks + 1;
11        end Finished;
12        procedure Reset is
13        begin
14            Finished_Tasks := 0;
15        end Reset;
16    end My_Barrier;
17    Phase_1, Phase_2: My_Barrier;
18    ... -- Phases_1 declaration
19    protected body Phases_1 is
20        entry Wait_1 (First_Pos, Last_Pos, K: out Integer; ToEnd: out Boolean)
21            when Local_Next_1 = True or Local_ToEnd = True is
22        begin
23            First_Pos := Local_First_Pos_1; Last_Pos := Local_Last_Pos_1;
24            K := Local_K_1; ToEnd := Local_ToEnd; Local_Next_1 := False;
25        end Wait_1;
26        procedure Start_1 (First_Pos, Last_Pos, K: Integer) is
27        begin
28            Local_First_Pos_1 := First_Pos; Local_Last_Pos_1 := Last_Pos;
29            Local_K_1 := K; Local_Next_1 := True;
30        end Start_1;
31        procedure Finished is
32        begin
33            Local_ToEnd := True;
34        end Finished;
35    end Phases_1;
36    ... -- Phases_2 declaration
37    protected body Phases_2 is
38        entry Wait_2 (First_Pos, Last_Pos, K, I: out Integer; ToEnd: out Boolean)
39            when Local_Next_2 = True or Local_ToEnd = True is
40        begin
41            First_Pos := Local_First_Pos_2; Last_Pos := Local_Last_Pos_2;
42            K := Local_K_2; I := Local_I; ToEnd := Local_ToEnd; Local_Next_2 := False;
43        end Wait_2;
44        procedure Start_2 (First_Pos, Last_Pos, K, I: Integer) is
45        begin
46            Local_First_Pos_2 := First_Pos; Local_Last_Pos_2 := Last_Pos;
47            Local_K_2 := K; Local_I := I; Local_Next_2 := True;
48        end Start_2;
49        procedure Finished is
50        begin
51            Local_ToEnd := True;
52        end Finished;
53    end Phases_2;
54
55    Process_Phases_1: array (0..Num_Tasks-1) of Phases_1;
56    Process_Phases_2: array (0..Num_Tasks-1) of Phases_2;
57    -- continues ...
```

Listing B.15: Cholesky kernel implemented with Ada tasks: synchronization mechanisms.

```
1   -- continues ...
2   ... -- Process_1 declaration
3   task body Process_1 is
4       Local_First_Pos, Local_Last_Pos, Local_K: Integer;
5       Local_Id : Integer;
6       ToEnd: Boolean;
7   begin
8       accept Id(My_Id: Integer) do
9           Local_Id := My_Id;
10      end Id;
11      loop
12          Process_Phases_1(Local_Id).wait_1(Local_First_Pos, Local_Last_Pos,
13                                            Local_K, ToEnd);
14          exit when ToEnd = True;
15          for I in Local_First_Pos .. Local_Last_Pos loop
16              Omp_trsm(M(Local_K, Local_K), M(Local_K, I), BS, BS);
17          end loop;
18          Phase_1.Finished;
19      end loop;
20  end Process_1;
21
22  ... -- Process_2 declaration
23  task body Process_2 is
24      Local_First_Pos, Local_Last_Pos, Local_K, Local_I: Integer;
25      Local_Id : Integer;
26      ToEnd: Boolean;
27  begin
28      accept Id(My_Id: Integer) do
29          Local_Id := My_Id;
30      end Id;
31      loop
32          Process_Phases_2(Local_Id).wait_2(Local_First_Pos, Local_Last_Pos,
33                                            Local_K, Local_I, ToEnd);
34          exit when ToEnd = True;
35          for J in Local_First_Pos .. Local_Last_Pos loop
36              Omp_gemm(M(Local_K, Local_I), M(Local_K, J), M(J, Local_I), BS, BS);
37          end loop;
38          Phase_2.Finished;
39      end loop;
40  end Process_2;
41
42  First_Pos, Last_Pos: Integer;
43  Temp_Size, Offset: Integer;
44  Process_Tasks_1: array (0..Num_Tasks-1) of Process_1;
45  Process_Tasks_2: array (0..Num_Tasks-1) of Process_2;
46  -- continues ...
```

Listing B.16: Cholesky kernel implemented with Ada tasks: secondary kernels.

```
1    -- continues ...
2 begin
3    for I in 0..Num_Tasks-1 loop
4       Process_Tasks_1(I).Id(I);
5       Process_Tasks_2(I).Id(I);
6    end loop;
7    for K in 0 .. NB - 1 loop
8       Omp_potrf(M(K, K), BS, BS);
9       Temp_Size := (NB-1) - (K + 1) + 1;
10      Offset := K +1;
11      If Temp_Size < Min_Chunk_Size * Num_Tasks then
12         for I in K + 1 .. NB-1 loop
13         Omp_trsm(M(K, K), M(K, I), BS, BS);
14         end loop;
15      else
16         for TI in 0..Num_Tasks-1 loop
17            First_Pos := Offset + TI * (Temp_Size / Num_Tasks);
18            Last_Pos := Offset + (TI+1) * (Temp_Size / Num_Tasks) - 1;
19            if TI = Num_Tasks-1 then
20               Last_Pos := Offset + Temp_Size - 1;
21            end if;
22            Process_Phases_1(TI).Start_1(First_Pos, Last_Pos, K);
23         end loop;
24         Phase_1.Wait;
25      end if;
26      for I in K + 1 .. NB-1 loop
27         If Temp_Size < Min_Chunk_Size * Num_Tasks then
28            for J in K + 1 .. i-1 loop
29               Omp_gemm(M(K, I), M(K, J), M(J, I), BS, BS);
30            end loop;
31         else
32            for TI in 0..Num_Tasks-1 loop
33               First_Pos := Offset + TI * (Temp_Size / Num_Tasks);
34               Last_Pos := Offset + (TI+1) * (Temp_Size / Num_Tasks) - 1;
35               if TI = Num_Tasks-1 then
36                  Last_Pos := Offset + Temp_Size - 1;
37               end if;
38               Process_Phases_2(TI).Start_2(First_Pos, Last_Pos, K, I);
39            end loop;
40            Phase_2.Wait;
41         end if;
42         Omp_syrk(M(K, I), M(I, I), BS, BS);
43      end loop;
44   end loop;
45
46   for I in 0..Num_Tasks-1 loop
47      Process_Phases_1(I).Finished;
48      Process_Phases_2(I).Finished;
49   end loop;
50 end cholesky_ada_tasks;
```

Listing B.17: Cholesky kernel implemented with Ada tasks: main kernel.

B.2.1.6 Ada + Paraffin

```
1 procedure cholesky_ada_paraffin (M : in out Matrix_Type) is
2     type Matrix_Dim is range 0 .. NB - 1;
3     package Parallel_Loops is new Parallel.Loops (Matrix_Dim);
4     package Iterate is new Parallel_Loops.Work_Sharing;
5 
6     Global_K , Global_I: Integer;
7     Num_Workers : Parallel.Worker_Count_Type := Parallel.Worker_Count_Type(Num_Tasks);
8     Min_Chunk_Size : Integer := 1;
9     Manager : Iterate.Work_Sharing_Manager := Iterate.Create ;
10    Temp_Size: Integer;
11
12    procedure Phase_1 (Start , Finish: Matrix_Dim) is
13        Local_Start: Integer := Integer(Start);
14        Local_Finish: Integer := Integer(Finish);
15    begin
16        for I in Local_Start .. Local_Finish loop
17            Omp_trsm(M(Global_K , Global_K), M(Global_K , I), BS, BS);
18        end loop;
19    end Phase_1;
20    procedure Phase_2 (Start , Finish: Matrix_Dim) is
21        Local_Start: Integer := Integer(Start);
22        Local_Finish: Integer := Integer(Finish);
23    begin
24        for J in Local_Start .. Local_Finish loop
25            Omp_gemm(M(Global_K , Global_I), M(Global_K , J), M(J , Global_I), BS, BS);
26        end loop;
27    end Phase_2;
28 begin
29    for K in 0 .. NB-1 loop
30        Global_K := K;
31        Omp_potrf(M(K, K), BS, BS);
32        Temp_Size := (NB-1) - (K + 1) + 1;
33        If Temp_Size < Min_Chunk_Size * Integer(Num_Workers) then
34            for I in K + 1 .. NB-1 loop
35                Omp_trsm(M(K, K), M(K, I), BS, BS);
36            end loop;
37        else
38            Manager.Execute_Parallel_Loop(Process => Phase_1'Access,
39                From => Matrix_Dim(K + 1), Worker_Count => Num_Workers);
40        end if;
41        for I in K + 1 .. NB-1 loop
42            Global_I := I;
43            If Temp_Size < Min_Chunk_Size * Integer(Num_Workers) then
44                for J in K + 1 .. i-1 loop
45                    Omp_gemm(M(K, I), M(K, J), M(J , I), BS, BS);
46                end loop;
47            else
48                Manager.Execute_Parallel_Loop(Process => Phase_2'Access,
49                    From => Matrix_Dim(K + 1), Worker_Count => Num_Workers);
50            end if;
51            Omp_syrk(M(K, I), M(I , I), BS, BS);
52        end loop;
53    end loop;
54 end cholesky_ada_paraffin;
```

Listing B.18: Cholesky kernel implemented with Paraffin.

B.2.2 LU factorization

B.2.2.1 C

```
1 void lu_c_sequential(double M[NB][NB][BS*BS]) {
2     for (int kk=0; kk<S; kk++) {
3         lu0(M[kk][kk]);
4         for (int jj=kk+1; jj<S; jj++)
5             fwd(M[kk][kk], M[kk][jj]);
6         for (int ii=kk+1; ii<S; ii++)
7             bdiv (M[kk][kk], M[ii][kk]);
8         for (int ii=kk+1; ii<S; ii++)
9             for (jj=kk+1; jj<S; jj++)
10                bmod(M[ii][kk], M[kk][jj], M[ii][jj]);
11    }
12 }
```

Listing B.19: LU kernel implemented with C.

B.2.2.2 C + OpenMP

```
1 void lu_c_task_taskwaits(struct parallel_data *args) {
2     float (**const M)[S][BS][BS] = &(*(*args).M);
3     if (GOMP_single_start())
4         for (int kk=0; kk<S; kk++) {
5             lu0((*M)[kk][kk]);
6             for (int jj=kk+1; jj<S; jj++) {
7                 struct fwd_data targs;
8                 targs.M = (float (**)[S][BS][BS]) M;
9                 targs.kk = kk; targs.jj = jj;
10                GOMP_task((void (*)(void *))fwd_task, &targs,
11                          (void (*)(void *, void *))0, 16, 8, 1,
12                          GOMP_TASK_UNTIED, 0, 0);
13            }
14            for (int ii=kk+1; ii<S; ii++) {
15                struct bdiv_data targs;
16                targs.M = (float (**)[S][BS][BS]) M;
17                targs.kk = kk; targs.ii = ii;
18                GOMP_task((void (*)(void *))bdiv_task, &targs,
19                          (void (*)(void *, void *))0, 16, 8, 1,
20                          GOMP_TASK_UNTIED, 0, 0);
21            }
22            GOMP_taskwait();
23            for (int ii=kk+1; ii<S; ii++)
24                for (int jj=kk+1; jj<S; jj++) {
25                    struct bmod_args targs;
26                    targs.M = (float (**)[S][BS][BS]) M;
27                    targs.kk = kk; targs.jj = jj; targs.ii = ii;
28                    GOMP_task((void (*)(void *))bmod_task, &targs,
29                              (void (*)(void *, void *))0, 24, 8, 1,
30                              GOMP_TASK_UNTIED, 0, 0);
31                }
32            GOMP_taskwait();
33        }
34 }
```

Listing B.20: LU kernel implemented with C and OpenMP tasks, and synchronizing tasks with taskwaits.

```
1 void lu_c_task_dependences(struct parallel_data *args) {
2    float (**const M)[S][BS][BS] = &(*(*args).M);
3    if (GOMP_single_start())
4       for (int kk=0; kk<S; kk++) {
5          struct deps_task_args targs;
6          targs.M = (float (**)[S][BS][BS]) M; targs.kk = kk;
7          void *deps[3L] = {[0] = (void *)1U, [1] = (void *)1U,
8                            [2] = (*M)[kk][kk]};
9          GOMP_task((void (*)(void *))lu0_task, &targs, (void (*)(void *, void *))0,
10                   16, 8, 1, UNTIED_DEPEND, deps, 0);
11         for (int jj=kk+1; jj<S; jj++) {
12            struct fwd_data targs;
13            targs.M = (float (**)[S][BS][BS]) M;
14            targs.kk = kk; targs.jj = jj;
15            void *deps[4L] = {[0] = (void *)2U, [1] = (void *)1U,
16                              [2] = (*M)[kk][kk], [3] = (*M)[kk][jj]};
17            GOMP_task((void (*)(void *))fwd_task, &targs, (void (*)(void *, void *))0,
18                      16, 8, 1, UNTIED_DEPEND, deps, 0);
19         }
20         for (int ii=kk+1; ii<S; ii++) {
21            struct deps_task_args targs;
22            targs.M = (float (**)[S][BS][BS]) M;
23            targs.kk = kk; targs.ii = ii;
24            void *deps[4L] = {[0] = (void *)2U, [1] = (void *)1U,
25                              [2] = (*M)[kk][kk], [3] = (*M)[ii][kk]};
26            GOMP_task((void (*)(void *))bdiv_task, &targs, (void (*)(void *, void *))0,
27                      16, 8, 1, UNTIED_DEPEND, deps, 0);
28         }
29         for (int ii=kk+1; ii<S; ii++)
30            for (int jj=kk+1; jj<S; jj++) {
31               struct deps_task_args targs;
32               targs.M = (float (**)[S][BS][BS]) M;
33               targs.kk = kk; targs.jj = jj; targs.ii = ii;
34               void *deps[5L] = {[0] = (void *)3U, [1] = (void *)1U, [2] = (*M)[ii][kk],
35                                 [3] = (*M)[kk][jj], [4] = (*M)[ii][jj]};
36               GOMP_task((void (*)(void *))bmod_task, &targs,
37                         (void (*)(void *, void *))0, 24, 8, 1, UNTIED_DEPEND, deps, 0);
38            }
39      }
40 }
```

Listing B.21: LU kernel implemented with C and OpenMP tasks, and synchronizing tasks with dependences.

B.2.2.3 Ada

```
1 procedure lu_ada_sequential(M : in out Matrix_Type) is
2 begin
3    for kk in 0 .. S-1 loop
4       Lu0(M(kk, kk));
5       for jj in kk+1 .. S-1 loop
6          Fwd(M(kk, kk), M([kk, jj));
7       for ii in kk+1 .. S-1 loop
8          Bdiv(M(kk, kk), M(ii, kk));
9       for ii in kk+1 .. S-1 loop
10         for jj in kk+1 .. S-1 loop
11            Bmod(M(ii, kk), M(kk, jj), M(ii, jj));
12   end loop;
13 end lu_ada_sequential;
```

Listing B.22: LU kernel implemented with Ada.

B.2.2.4 Ada + OpenMP

```
procedure lu_ada_task_taskwaits(args: System.Address) is
   ... -- Declarations
   args_Acc: Parallel_Data_Type_Access := Convert_to_Parallel(args);
   M_Acc: Matrix_Access := Convert_to_Matrix(args_Acc.M_Address);
   Depend_Clauses: OpenMP.Void_Ptr_Ptr := null;
begin
   if (OpenMP.Ada_OMP_Get_Thread_Num = 0) then
      for KK in 0 .. S-1 loop
         Lu0(M_Acc.all(KK, KK));
         for JJ in KK + 1 .. S-1 loop
            ... -- Declarations
            begin
               targs.M_Address := args_Acc.M_Address;
               targs.KK := KK; targs.JJ := JJ;
               targs := targs.all'Address;
               OpenMP.Ada_GOMP_Task(Fwd_task'Unrestricted_Access,
                                    targs.all'Address, null, 16, 8, TRUE,
                                    GOMP_TASK_UNTIED, Depend_Clauses, 0);
            end;
         end loop;
         for II in KK + 1 .. S -1 loop
            ... -- Declarations
            begin
               targs.M_Address := args_Acc.M_Address;
               targs.KK := KK; targs.II := II;
               OpenMP.Ada_GOMP_Task(Bdiv_task'Unrestricted_Access,
                                    targs.all'Address, null, 16, 8, TRUE,
                                    GOMP_TASK_UNTIED, Depend_Clauses, 0);
            end;
         end loop;
         OpenMP.Ada_GOMP_Taskwait;
         for II in KK + 1 .. S -1 loop
            for JJ in KK + 1 .. S-1 loop
               ... -- Declarations
               begin
                  targs.M_Address := args_Acc.M_Address;
                  targs.KK := KK; targs.JJ := JJ; targs.II := II;
                  OpenMP.Ada_GOMP_Task(Bmod_task'Unrestricted_Access,
                                       targs.all'Address, null, 20, 8, TRUE,
                                       GOMP_TASK_UNTIED, Depend_Clauses, 0);
               end;
            end loop;
         end loop;
         OpenMP.Ada_GOMP_Taskwait;
      end loop;
  end if;
end lu_ada_task_taskwaits;
```

Listing B.23: LU kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with taskwaits.

```
1 procedure lu_ada_task_dependences(args: System.Address) is
2     ... -- Declarations
3 begin
4    if (OpenMP.Ada_OMP_Get_Thread_Num = 0) then
5       for KK in 0 .. S-1 loop
6          declare
7             targs : Lu0_Access_Type := new Lu0_Type;
8             Depend_Clauses_Array: aliased array(1..3) of aliased OpenMP.Void_Ptr :=
9                (DEPS_PTR_1U, DEPS_PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(kk,kk)));
10         begin
11            targs.M_Address := args_Acc.M_Address; targs.KK := KK;
12            OpenMP.Ada_GOMP_Task(Tasks_Function_Task_1'Unrestricted_Access,
13                 targs.all'Address, null, 12, 8, TRUE, UNTIED_DEPEND,
14                 Depend_Clauses_Array(1)'Unrestricted_Access, 0);
15         end;
16         for JJ in KK + 1 .. S-1 loop
17            declare
18               targs: Fwd_Access_Type := new Fwd_Type;
19               Depend_Clauses_Array: aliased array(1..4) of aliased OpenMP.Void_Ptr:=
20                  (DEPS_PTR_2U, DEPS_PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(kk,kk)),
21                  Submatrix_to_Void_Ptr(M_Acc.all(kk,jj)));
22            begin
23               targs.M_Address := args_Acc.M_Address; targs.KK := KK; targs.JJ := JJ;
24               OpenMP.Ada_GOMP_Task(Tasks_Function_Task_2'Unrestricted_Access,
25                  targs.all'Address, null, 16, 8, TRUE, UNTIED_DEPEND,
26                  Depend_Clauses_Array(1)'Unrestricted_Access, 0);
27            end;
28         end loop;
29         for II in KK + 1 .. S -1 loop
30            declare
31               targs: Bdiv_Access_Type := new Bdiv_Type;
32               Depend_Clauses_Array: aliased array(1..4) of aliased OpenMP.Void_Ptr:=
33                  (DEPS_PTR_2U, DEPS_PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(kk,kk)),
34                  Submatrix_to_Void_Ptr(M_Acc.all(ii,kk)));
35            begin
36               targs.M_Address := args_Acc.M_Address; targs.KK := KK; targs.II := II;
37               OpenMP.Ada_GOMP_Task(Tasks_Function_Task_3'Unrestricted_Access,
38                  targs.all'Address, null, 16, 8, TRUE, UNTIED_DEPEND,
39                  Depend_Clauses_Array(1)'Unrestricted_Access, 0);
40            end;
41         end loop;
42         for II in KK + 1 .. S -1 loop
43            for JJ in KK + 1 .. S-1 loop
44               declare
45                  targs: Bmod_Access_Type := new Bmod_Type;
46                  Depend_Clauses_Array: aliased array(1..5) of aliased OpenMP.Void_Ptr:=
47                     (DEPS_PTR_3U, DEPS_PTR_1U, Submatrix_to_Void_Ptr(M_Acc.all(ii,kk)),
48                     Submatrix_to_Void_Ptr(M_Acc.all(kk,jj)),
49                     Submatrix_to_Void_Ptr(M_Acc.all(ii,jj)));
50               begin
51                  targs.M_Address := args_Acc.M_Address;
52                  targs.KK := KK; targs.JJ := JJ; targs.II := II;
53                  OpenMP.Ada_GOMP_Task(Tasks_Function_Task_4'Unrestricted_Access,
54                       targs.all'Address, null, 20, 8, TRUE, UNTIED_DEPEND,
55                       Depend_Clauses_Array(1)'Unrestricted_Access, 0);
56               end;
57            end loop;
58         end loop;
59      end loop;
60   end if;
61 end lu_ada_task_dependences;
```

Listing B.24: LU kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with dependences.

B.2.2.5 Ada tasks

```
1 procedure lu_ada_tasks(M : in out Matrix_Type) is
2     Num_Tasks: Integer := Num_Tasks;
3     Min_Chunk_Size : Integer := 1;
4     Size: Integer := S;
5 
6     protected type My_Barrier is
7         entry Wait;
8         procedure Finished;
9         procedure Reset;
10    private
11        Finished_Tasks : Integer := 0;
12        N_Tasks: Integer := Num_Tasks;
13    end My_Barrier;
14    protected body My_Barrier is
15        entry Wait when Finished_Tasks = N_Tasks is
16        begin
17            Finished_Tasks := 0;
18        end;
19        procedure Finished is
20        begin
21            Finished_Tasks := Finished_Tasks + 1;
22        end Finished;
23        procedure Reset is
24        begin
25            Finished_Tasks := 0;
26        end Reset;
27    end My_Barrier;
28 
29    Phase_1, Phase_2, Phase_3: My_Barrier;
30    -- continues ...
```

Listing B.25: LU kernel implemented with Ada tasks: synchronization mechanisms.

```
1  -- continues ...
2  protected body Phases is
3      entry Wait_1 (First_Pos, Last_Pos, KK: out Integer; ToEnd: out Boolean)
4      when Local_Next_1 = True or Local_ToEnd = True is
5      begin
6          First_Pos := Local_First_Pos_1; Last_Pos := Local_Last_Pos_1;
7          KK := Local_KK_1; ToEnd := Local_ToEnd; Local_Next_1 := False;
8      end Wait_1;
9      procedure Start_1 (First_Pos, Last_Pos, KK: Integer) is
10     begin
11         Local_First_Pos_1 := First_Pos; Local_Last_Pos_1 := Last_Pos;
12         Local_KK_1 := KK; Local_Next_1 := True;
13     end Start_1;
14
15     entry Wait_2 (First_Pos, Last_Pos, KK: out Integer; ToEnd: out Boolean)
16     when Local_Next_2 = True or Local_ToEnd = True is
17     begin
18         First_Pos := Local_First_Pos_2; Last_Pos := Local_Last_Pos_2;
19         KK := Local_KK_2; ToEnd := Local_ToEnd; Local_Next_2 := False;
20     end Wait_2;
21     procedure Start_2 (First_Pos, Last_Pos, KK: Integer) is
22     begin
23         Local_First_Pos_2 := First_Pos; Local_Last_Pos_2 := Last_Pos;
24         Local_KK_2 := KK; Local_Next_2 := True;
25     end Start_2;
26
27     entry Wait_3 (First_Pos, Last_Pos, KK: out Integer; ToEnd: out Boolean)
28     when Local_Next_3 = True or Local_ToEnd = True is
29     begin
30         First_Pos := Local_First_Pos_3; Last_Pos := Local_Last_Pos_3;
31         KK := Local_KK_3; ToEnd := Local_ToEnd; Local_Next_3 := False;
32     end Wait_3;
33     procedure Start_3 (First_Pos, Last_Pos, KK: Integer) is
34     begin
35         Local_First_Pos_3 := First_Pos; Local_Last_Pos_3 := Last_Pos;
36         Local_KK_3 := KK; Local_Next_3 := True;
37     end Start_3;
38
39     procedure Finished is
40     begin
41         Local_ToEnd := True;
42     end Finished;
43 end Phases;
44
45 Process_Phases: array (0..Num_Tasks-1) of Phases;
46 -- continues ...
```

Listing B.26: LU kernel implemented with Ada tasks: secondary kernels.

```
-- continues ...
... -- Task declaration
task body Process is
    ... -- Declarations
begin
    accept Id(My_Id: Integer) do
        Local_Id := My_Id;
    end Id;

    loop
        Process_Phases(Local_Id).wait_1(Local_First_Pos, Local_Last_Pos,
                                        Local_KK, ToEnd);
        exit when ToEnd = True;
        for JJ in Local_First_Pos .. Local_Last_Pos loop
            Changed_Fwd(M, Local_KK, JJ);
        end loop;
        Process_Phases(Local_Id).wait_2(Local_First_Pos, Local_Last_Pos,
                                        Local_KK, ToEnd);
        exit when ToEnd = True;
        for II in Local_First_Pos .. Local_Last_Pos loop
            Changed_Bdiv(M, Local_KK, II);
        end loop;
        Phase_2.Finished;

        Process_Phases(Local_Id).wait_3(Local_First_Pos, Local_Last_Pos,
                                        Local_KK, ToEnd);
        exit when ToEnd = True;

        for II in Local_First_Pos .. Local_Last_Pos loop
            for JJ in Local_KK + 1 .. S -1 loop
                Changed_Bmod(M, Local_KK, II, JJ);
            end loop;
        end loop;
        Phase_3.Finished;
    end loop;
end Process;

First_Pos, Last_Pos: Integer;
Process_Tasks: array (0..Num_Tasks-1) of Process;
Temp_Size, Offset: Integer;
-- continues ...
```

Listing B.27: LU kernel implemented with Ada tasks: more secondary kernels.

```
1      -- continues ...
2  begin
3      for I in 0..Num_Tasks-1 loop
4          Process_Tasks(I).Id(I);
5      end loop;
6
7      for KK in 0 .. S - 1 loop
8          Changed_Lu0(M, KK);
9
10         Temp_Size := (S-1) - (KK + 1) + 1;
11         Offset := KK + 1;
12
13         If Temp_Size < Min_Chunk_Size * Num_Tasks then
14             for JJ in KK+1 .. S -1 loop
15                 Changed_Fwd(M, KK, JJ);
16             end loop;
17             for II in KK+1 .. S-1 loop
18                 Changed_Bdiv(M, KK, II);
19             end loop;
20             for II in KK +1 .. S -1 loop
21                 for JJ in KK + 1 .. S -1 loop
22                     Changed_Bmod(M, KK, II , JJ);
23                 end loop;
24             end loop;
25         else
26             for I in 0..Num_Tasks-1 loop
27                 First_Pos := Offset + I * (Temp_Size / Num_Tasks);
28                 Last_Pos := Offset + (I+1) * (Temp_Size / Num_Tasks) - 1;
29                 if I = Num_Tasks-1 then
30                     Last_Pos := Offset + Temp_Size - 1;
31                 end if;
32                 Process_Phases(I).Start_1(First_Pos , Last_Pos , KK);
33             end loop;
34             for I in 0..Num_Tasks-1 loop
35                 First_Pos := Offset + I * (Temp_Size / Num_Tasks);
36                 Last_Pos := Offset + (I+1) * (Temp_Size / Num_Tasks) - 1;
37                 if I = Num_Tasks-1 then
38                     Last_Pos := Offset + Temp_Size - 1;
39                 end if;
40                 Process_Phases(I).Start_2(First_Pos , Last_Pos , KK);
41             end loop;
42             Phase_2.Wait;
43
44             for I in 0..Num_Tasks-1 loop
45                 First_Pos := Offset + I * (Temp_Size / Num_Tasks);
46                 Last_Pos := Offset + (I+1) * (Temp_Size / Num_Tasks) - 1;
47                 if I = Num_Tasks-1 then
48                     Last_Pos := Offset + Temp_Size - 1;
49                 end if;
50                 Process_Phases(I).Start_3(First_Pos , Last_Pos , KK);
51             end loop;
52             Phase_3.Wait;
53         end if;
54     end loop;
55     for I in 0..Num_Tasks-1 loop
56         Process_Phases(I).Finished;
57     end loop;
58 end lu_ada_tasks;
```

Listing B.28: LU kernel implemented with Ada tasks: main kernel.

B.2.2.6 Ada + Paraffin

```
1 procedure lu_ada_paraffin (M : in out Matrix_Type) is
2     type Matrix_Dim is range 0 .. S - 1;
3     Min_Chunk_Size : Integer := 1;
4     Num_Workers : Parallel.Worker_Count_Type := Parallel.Worker_Count_Type(Num_Tasks);
5     Global_KK: Integer := 0;
6     Temp_Size: Integer;
7     package Parallel_Loops is new Parallel.Loops (Matrix_Dim);
8     package Iterate is new Parallel_Loops.Work_Sharing;
9     Manager : Iterate.Work_Sharing_Manager := Iterate.Create ;
10
11    procedure Phase_1_2 (Start , Finish: Matrix_Dim) is
12        Local_Start: Integer := Integer(Start); Local_Finish: Integer := Integer(Finish);
13    begin
14        for JJ in Local_Start .. Local_Finish loop
15            Changed_Fwd(M, Global_KK , JJ);
16        end loop;
17        for II in Local_Start .. Local_Finish loop
18            Changed_Bdiv(M, Global_KK , II);
19        end loop;
20    end Phase_1_2;
21
22    procedure Phase_3 (Start , Finish: Matrix_Dim) is
23        Local_Start: Integer := Integer(Start); Local_Finish: Integer := Integer(Finish);
24    begin
25        for II in Local_Start .. Local_Finish loop
26            for JJ in Global_KK + 1 .. S -1 loop
27                Changed_Bmod(M, Global_KK , II , JJ);
28            end loop;
29        end loop;
30    end Phase_3;
31 begin
32    for KK in 0 .. S - 1 loop
33        Changed_Lu0(M, KK);
34        Global_KK := KK;
35        Temp_Size := (S-1) - (KK + 1) + 1;
36
37        If Temp_Size < Min_Chunk_Size * Integer(Num_Workers) then
38            for JJ in KK+1 .. S -1 loop
39                Changed_Fwd(M, KK, JJ);
40            end loop;
41            for II in KK+1 .. S-1 loop
42                Changed_Bdiv(M, KK, II);
43            end loop;
44            for II in KK +1 .. S -1 loop
45                for JJ in KK + 1 .. S -1 loop
46                    Changed_Bmod(M, KK, II , JJ);
47                end loop;
48            end loop;
49        else
50            Manager.Execute_Parallel_Loop(Process => Phase_1_2'Access,
51                From => Matrix_Dim(KK + 1), Worker_Count => Num_Workers);
52
53            Manager.Execute_Parallel_Loop(Process => Phase_3'Access,
54                From => Matrix_Dim(KK + 1), Worker_Count => Num_Workers);
55        end if;
56    end loop;
57 end lu_ada_paraffin;
```

Listing B.29: LU kernel implemented with Ada and Paraffin: synchronization mechanisms.

B.2.3 Matrix

B.2.3.1 C

```
1 void matrix_computation(float* A)
2 {
3     float res = 0;
4     for (int i=1; i<=Simul_Load; ++i)
5         res += *A * 2.0;
6     *A = res;
7 }
8 void matrix_c_seq(float ***M)
9 {
10    for (int i=0; i<Size; ++i)
11        for (int j=0; j<Size; ++j)
12            matrix_computation(&((*M)[i][j]));
13 }
```

Listing B.30: Matrix kernel implemented with C.

B.2.3.2 C + OpenMP

```
1 void matrix_computation_task(struct matrix_task* args)
2 {
3     for (int i=args->First_Pos; i<=args->Last_Pos; ++i)
4         for (int j=0; j<Size; ++j)
5             matrix_computation(&((*args->M)[i][j]));
6 }
7 void matrix_c_task_taskwaits (float ***M)
8 {
9     if (omp_get_thread_num() == 0) {
10        int First_Pos, Last_Pos;
11        for (int K=0; K<n_tasks; ++K)
12        {
13            First_Pos = K * (Size / n_tasks);
14            Last_Pos = (K+1) * (Size / n_tasks) - 1;
15            if (K == n_tasks-1)
16                Last_Pos = Size-1;
17
18            struct matrix_data targs;
19            targs.M = args->M;
20            targs.First_Pos = First_Pos;
21            targs.Last_Pos = Last_Pos;
22            GOMP_task((void (*)(void *))matrix_computation_task,
23                      &targs, 0, 16, 4, 1, 0, 0, 0);
24        }
25        GOMP_taskwait();
26    }
27 }
```

Listing B.31: Matrix kernel implemented with C and OpenMP tasks, and synchronizing tasks with taskwaits.

B.2.3.3 Ada

```
1 procedure matrix_computation (A: in out Float) is
2     Res: Float := 0.0;
3 begin
4     for I in 1 .. Simul_Load loop
5         Res := Res + A * 2.0;
6     end loop;
7     A := Res;
8 end matrix_computation_task;
9
10 procedure matrix_ada_seq (M : in out Matrix) is
11 begin
12     for I in Matrix_Dim loop
13         for J in Matrix_Dim loop
14           matrix_computation_task(M(I, J));
15         end loop;
16     end loop;
17 end Process_Seq;
```

Listing B.32: Matrix kernel implemented with Ada.

B.2.3.4 Ada + OpenMP

```
1 procedure matrix_computation_task(args: System.Address) is
2     ... -- Declarations
3 begin
4     for I in args_Acc.First_Pos .. args_Acc.Last_Pos loop
5         for J in Matrix_Dim loop
6             matrix_computation(M_Access.all(Matrix_Dim(I), Matrix_Dim(J)));
7         end loop;
8     end loop;
9 end matrix_computation_task;
10
11 procedure matrix_ada_task_taskwaits (M : in out Matrix) is
12     Depend_Clauses: OpenMP.Void_Ptr_Ptr := null;
13 begin
14     for K in 0 .. Num_Tasks-1 loop
15         ... -- Declarations
16         begin
17             First_Pos := K * (Size / Num_Tasks) + 1;
18             Last_Pos := (K+1) * (Size / Num_Tasks);
19             targs.M_Address := M'Address;
20             targs.First_Pos := First_Pos; targs.Last_Pos := Last_Pos;
21             OpenMP.Ada_GOMP_Task(matrix_computation_task'Unrestricted_Access,
22                 targs.all'Address, null, 16, 4, TRUE, 0, Depend_Clauses, 0);
23         end;
24     end loop;
25   OpenMP.Ada_GOMP_Taskwait;
26 end matrix_ada_task_taskwaits;
```

Listing B.33: Matrix kernel implemented with Ada and OpenMP tasks, and synchronizing tasks with taskwaits.

B.2.3.5 Ada tasks

```
1 procedure matrix_ada_tasks(M : in out Matrix) is
2 ... -- Process declaration
3 task body Process is
4     Local_First_Pos, Local_Last_Pos: Positive;
5 begin
6     accept Start (First_Pos, Last_Pos: Positive) do
7         Local_Last_Pos := Last_Pos;
8         Local_First_Pos := First_Pos;
9     end Start;
10    for I in Local_First_Pos .. Local_Last_Pos loop
11        for J in Matrix_Dim loop
12            matrix_computation(M(Matrix_Dim(I), Matrix_Dim(J)));
13        end loop;
14    end loop;
15 end Process;
16
17 First_Pos, Last_Pos: Positive;
18 Process_Tasks: array (1..Num_Tasks) of Process;
19
20 begin
21 for I in 0..Num_Tasks-1 loop
22     First_Pos := I * (Size / Num_Tasks)+1;
23     Last_Pos := (I+1) * (Size / Num_Tasks);
24     if I = Num_Tasks then
25         Last_Pos := Integer(Matrix_Dim'Last);
26     end if;
27     Process_Tasks(I+1).Start(First_Pos, Last_Pos);
28 end loop;
29 end matrix_ada_tasks;
```

Listing B.34: Matrix kernel implemented with Ada tasks.

B.2.3.6 Ada + Paraffin

```
1 procedure matrix_ada_paraffin(M : in out Matrix_Type) is
2    package Parallel_Loops is new Parallel.Loops (Matrix_Dim);
3    package Iterate is new Parallel_Loops.Work_Sharing;
4
5    procedure Generic_Iterate (Start, Finish: Matrix_Dim; Row: Matrix_Dim) is
6    begin
7       for I in Start..Finish loop
8          matrix_computation(M(Row, I));
9       end loop;
10   end Generic_Iterate;
11
12   procedure Process_Row(Row: Matrix_Dim) is
13      Manager : Iterate.Work_Sharing_Manager := Iterate.Create ;
14      function Convert_to_WorkerCount is new Ada.Unchecked_Conversion(
15         Source=>Integer, Target=>Worker_Count_Type);
16      Nthreads_WC : Worker_Count_Type := Convert_to_WorkerCount(Nthreads);
17
18      procedure Iteration(Start, Finish: Matrix_Dim) is
19      begin
20         Generic_Iterate(Start, Finish, Row);
21      end Iteration;
22   begin
23      Manager.Execute_Parallel_Loop
24         (Process => Iteration'Access, Worker_Count => Nthreads_WC);
25   end Process_Row;
26 begin
27   for I in Matrix_Dim loop
28      Process_Row(I);
29   end loop;
30 end matrix_ada_paraffin;
```

Listing B.35: Matrix kernel implemented with Paraffin.

B.2.4 Synthetic: Ada tasks + OpenMP tasks

```
1 ... -- Task Declaration
2 task body Periodic is
3     T0: Time;
4     Period: Time_Span := Milliseconds(200);
5     Next: Time;
6 begin
7     T0   := Clock;
8     Next := T0 + Period;
9     for i in 1..50 loop
10        delay until Next;
11        Extrae.Ada_Extrae_event(6000, 1);
12        for i in 1..1000000 loop
13            Counter.Inc;
14        end loop;
15        Next := Next + Period;
16        Extrae.Ada_Extrae_event(6000, 0);
17    end loop;
18 end Periodic;
19
20 ... -- Protected Object Declaration
21 protected body Event is
22    procedure Release is
23    begin
24        Open := True;
25    end Release;
26    entry Wait when Open = True is
27    begin
28        Extrae.Ada_Extrae_event(6000, 2);
29        for i in 1..10000000 loop
30            Counter.Inc;
31        end loop;
32        Open := False;
33        Extrae.Ada_Extrae_event(6000, 0);
34    end Wait;
35 end Event;
36
37 ... -- Task Declaration
38 task body Sporadic is
39 begin
40    for I in 1..2 loop
41        Sporadic_Event.Wait;
42    end loop;
43 end Sporadic;
44
45 Sporadic_Event: Event;
```

Listing B.36: Synthetic kernel with interaction between Ada tasks and OpenMP tasks, using Extrae instrumentation tool: Ada tasks.

```
1 procedure synthetic_omp_task_1(args: System.Address) is
2     ... -- Declarations
3 begin
4     Extrae.Ada_Extrae_event(6000, 3);
5     for I in args_Acc.First_Pos .. args_Acc.Last_Pos loop
6         for J in Matrix_Dim loop
7         matrix_computation(M_Access.all(Matrix_Dim(I), Matrix_Dim(J)));
8         end loop;
9     end loop;
10    Counter.Inc;
11    Extrae.Ada_Extrae_event(6000, 0);
12 end synthetic_omp_task;
13
14 procedure synthetic_omp_task_2(Task_Params: System.Address) is
15    ... -- Declarations
16 begin
17    Extrae.Ada_Extrae_event(6000, 4);
18    for i in 1..100000000 loop
19        Res := Res + 1;
20    end loop;
21    Extrae.Ada_Extrae_event(6000, 0);
22 end synthetic_omp_task_2;
23
24 procedure synthetic_omp_task(M : in out Matrix) is
25    Depend_Clauses: OpenMP.Void_Ptr_Ptr := null;
26 begin
27   for K in 0 .. Num_Tasks-1 loop
28      ... -- Declarations
29      begin
30          First_Pos := K * (Size / Num_Tasks) + 1;
31          Last_Pos := (K+1) * (Size / Num_Tasks);
32          if K = Num_Tasks then
33              Last_Pos := Integer(Size);
34          end if;
35
36          targs.M_Address := M'Address;
37          targs.First_Pos := First_Pos;
38          targs.Last_Pos := Last_Pos;
39          targs := Task_Data_Access.all'Address;
40          OpenMP.Ada_GOMP_Task(synthetic_omp_task_1'Unrestricted_Access,
41              targs, null, 16, 4, TRUE, 0, Depend_Clauses, 0);
42          OpenMP.Ada_GOMP_Task(synthetic_omp_task_2'Unrestricted_Access,
43              targs, null, 16, 4, TRUE, 0, Depend_Clauses, 0);
44      end;
45   end loop;
46   OpenMP.Ada_GOMP_Taskwait;
47 end synthetic_omp_task;
```

Listing B.37: Synthetic kernel with interaction between Ada tasks and OpenMP tasks, using Extrae instrumentation tool: OpenMP tasks.

```ada
... -- Protected Object Declaration
protected body Counter is
    procedure Inc is
    begin
        Count := Count+1;
    end Inc;
    function Read return Natural is
    begin
        return Count;
    end Read;
end Counter;

procedure synthetic_main(Parallel_Params: System.Address) is
    ... -- Declarations
begin
    if (OpenMP.Ada_OMP_Get_Thread_Num = 0) then
        synthetic_omp_task(M_Access.all);
        Sporadic_Event.Release;
    elsif OpenMP.Ada_OMP_Get_Thread_Num = 1 then
        Sporadic_Event.Release;
    end if;
    for i in 1..1000000 loop
        Counter.Inc;
    end loop;
end synthetic_main;
```

Listing B.38: Synthetic kernel with interaction between Ada tasks and OpenMP tasks, using Extrae instrumentation tool: global objects and main function.

Acronyms

API Application Program Interface
AST Abstract Syntax Tree
BSC Barcelona Supercomputing Center
CFG Control Flow Graph
DAG Directed Acyclic Graph
DTO Data Transfer Object
HPC High-Performance Computing
IPA Interprocedural Analysis
IR Intermediate Representation
MPI Message Passing Interface
MPSoC Multiprocessor System-on-Chip
NUMA Non-Unified Memory Access
OpenMP Open Multi-Processing
OpenCL Open Computing Language
PCFG Parallel Control Flow Graph
SMP Symmetric Multi-Processing
SSA Static Single Assignment
TBB Threading Building Blocks
TDG Task Dependency Graph
TSC Task Scheduling Constraint
TSP Task Scheduling Point
VLIW Very Long Instruction Word